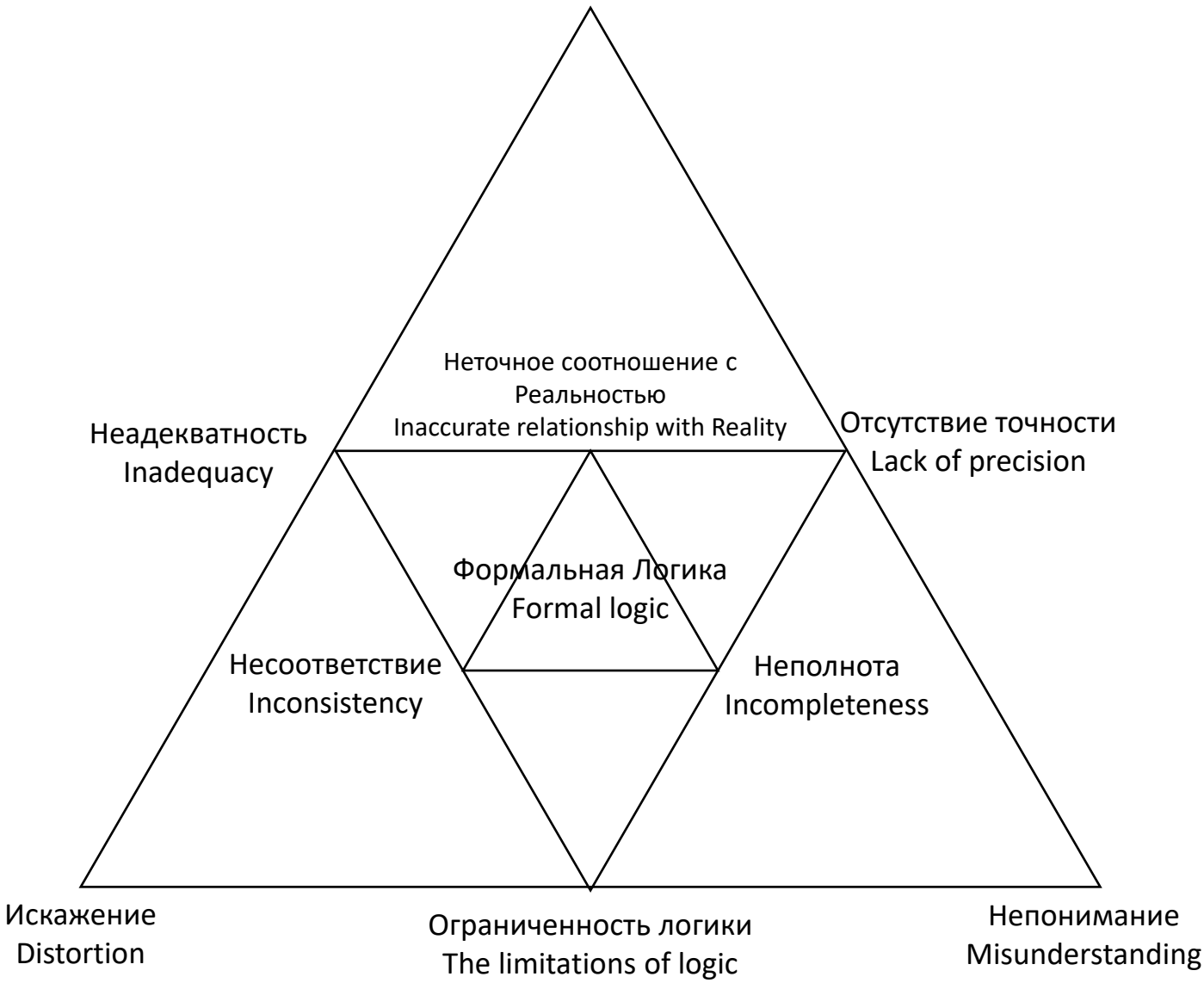
Надёжность и эффективность анализа.
Reliability and efficiency of analysis.



Adding an intermediate value (0.1) ("formal inference value")

1. The problem of incompleteness is solved by **10-20%**

2. The problem of inconsistency (inconsistency) is solved by **30%**

3. The problem of semantics is solved **20-40%**

Reliability and efficiency of analysis up to **+15-25%**

Example: 1.1 - 20% probability 0.1 - 30% probability 0.0 - 40% probability

Reliability and efficiency of analysis up to +50-70%

Федорченко Михаил Валерьевич

Example:
1.1.1.1.1 - 20% probability + reason
0.1.1.1.1 - 30% probability + reason
0.0.1.1.1 - 50% probability + reason
0.0.0.1.1 - 65% probability + reason
0.0.0.0.1 - 35% probability + reason
0.0.0.0.0 - 40% probability + reason

Advantages of the approach:

Maximum granularity: the system takes into account several dimensions of the conclusions.

Flexibility: levels can be adapted to different areas of knowledge.

Handling uncertainty: probabilities reflect the degree of confidence in the conclusions.

Potential problem reduction:

Incompleteness: up to 70–80%, due to the ability to work with intermediate states.

Inconsistency: up to 60–70%, due to the division into dimensions and the probabilistic approach.

Semantics: up to 80–90%, since the values have a clear multidimensional representation.

Reliability and efficiency of analysis up to +65-85%

Федорченко Михаил Валерьевич

Explanation of the structure:
Five-level structure (A.B.C.D.E):

(A): True/False

(1): True.
(0): False.
(B): Formality of inference

(1): Formal inference.
(0): Informal inference.
(C): Confidence (reliability)

(1): Reliable statement.
(0): Unreliable statement.
(D): Contextual relevance

(1): Consistent with the current context.
(0): Inconsistent.
(E): Model agreement

(1): Consistent with the model.
(0): Inconsistent with the model.

(G): Reason

(1): Valid.
(0): Invalid.

(1.1.1.1.1) (20%):

Complete truth, formal inference, reliability, contextual relevance, agreement with the model. Highest degree of confidence.
(0.1.1.1.1) (30%):G

False, but the inference is formal and reliable, with relevance and agreement with the model. An example of a situation where a false statement is still consistent with formal logic.
(0.0.1.1.1) (50%): G

False statement, non-formal inference, but reliable, contextually relevant and consistent with the model.
(0.0.0.1.1) (65%):G

False statement, non-formal inference, unreliable, but relevant and consistent.
(0.0.0.0.1) (35%):G

False statement, unformal, unreliable, irrelevant, but consistent.
(0.0.0.0.0) (40%):G

Totally false, unformal, unreliable, irrelevant, and inconsistent.

Федорченко Михаил Валерьевич

0.0.0.0 - 0.0.0.1 - 0.0.1.1 - 0.1.1.1 - 1.1.1.1. Пятый элемент логики - Мудрость, но Мудрый человек дальше квантовой логики, для ИИ, не пойдёт. Они хотят Супер ИИ, моя цель создать из Хомо Сапиенс - Мудрого Хомо. Это Спасение Мира и Жизни на Планете Земля.

0.0.0.0 - 0.0.0.1 - 0.0.1.1 - 0.1.1.1 - 1.1.1.1. The fifth element of logic is Wisdom, but the Wise Man will not go further than quantum logic for AI. They want Super AI, my goal is to create from Homo Sapiens - Wise Homo. This is the Salvation of World and Life on Planet Earth.

0-false **1**-true

0.0-false **0.1**-formal **1.1**-true

0.0.0-false **0.0.1**-subjectivity **0.1.1**-objectivity **1.1.1**-true

0.0.0.0-false**0.0.0.1**-subjectivity**0.0.1.1**-formal**0.1.1.1**-Wise**1.1.1.1**-true

0.0.0.0.0-false
0.0.0.0.1-subjectivity
0.0.0.1.1-formal
0.0.1.1.1-objectivity
0.1.1.1.1-Wise
1.1.1.1.1-true

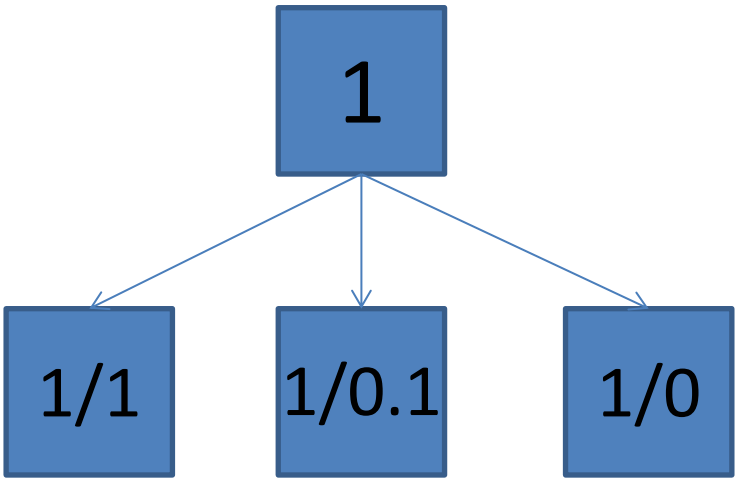
Add a third axis (e.g. confidence in the truth of T as a probability)

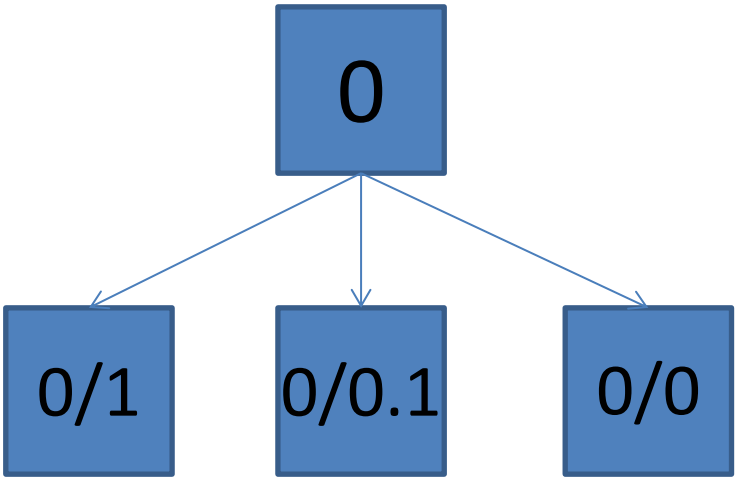
Y-axis: True (T=1)

1/1 True (Green) | 1/0.1 Almost True (Yellow) | 1/0 More True than False (Red) |
|------------------|
X-axis: False (T=0)

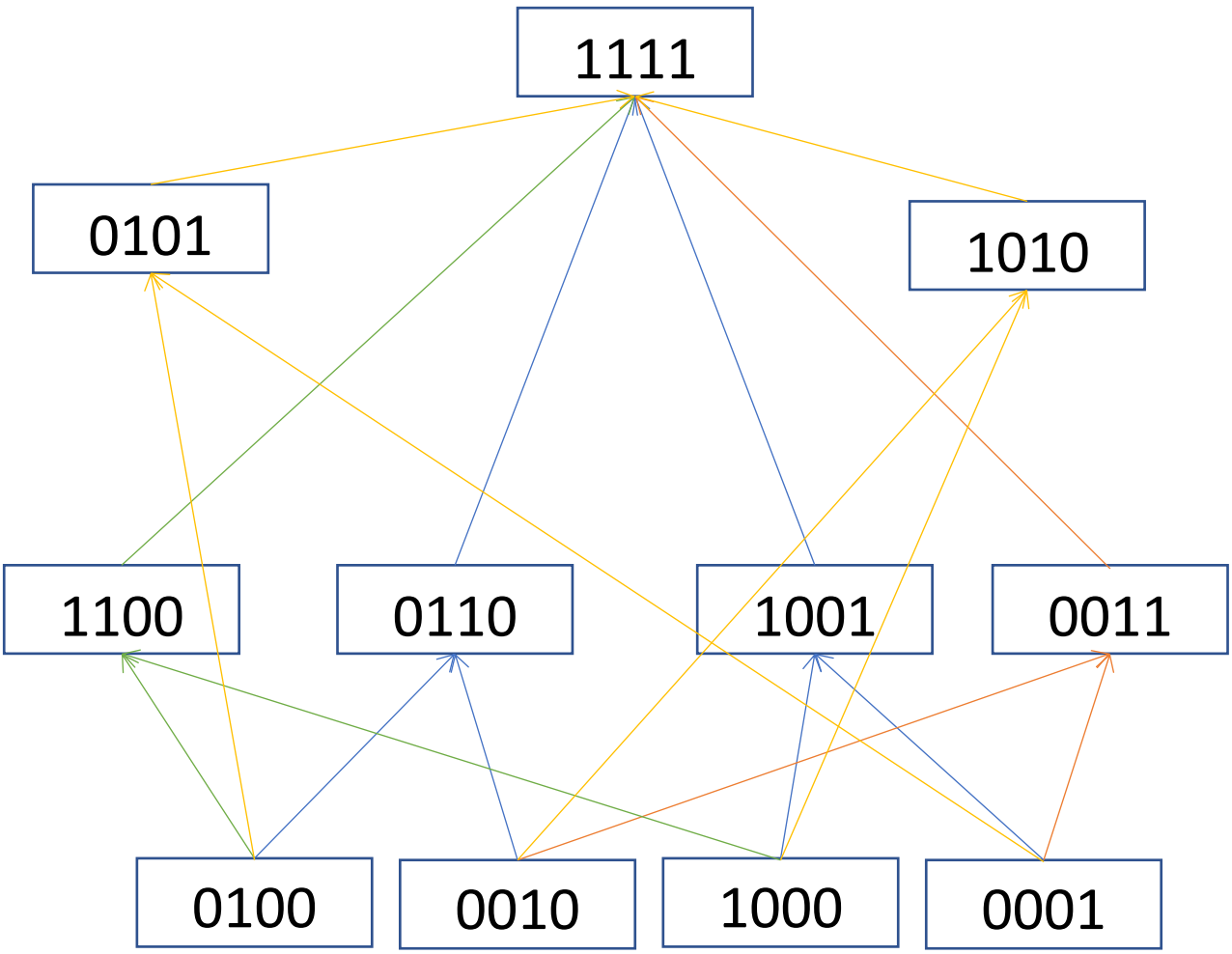
0/1 More False than True (Orange) | 0/0.1 Almost False (Red) | 0/0 False (Dark Red)|
|------------------|

Fedorchenko Mikhail Valerevich

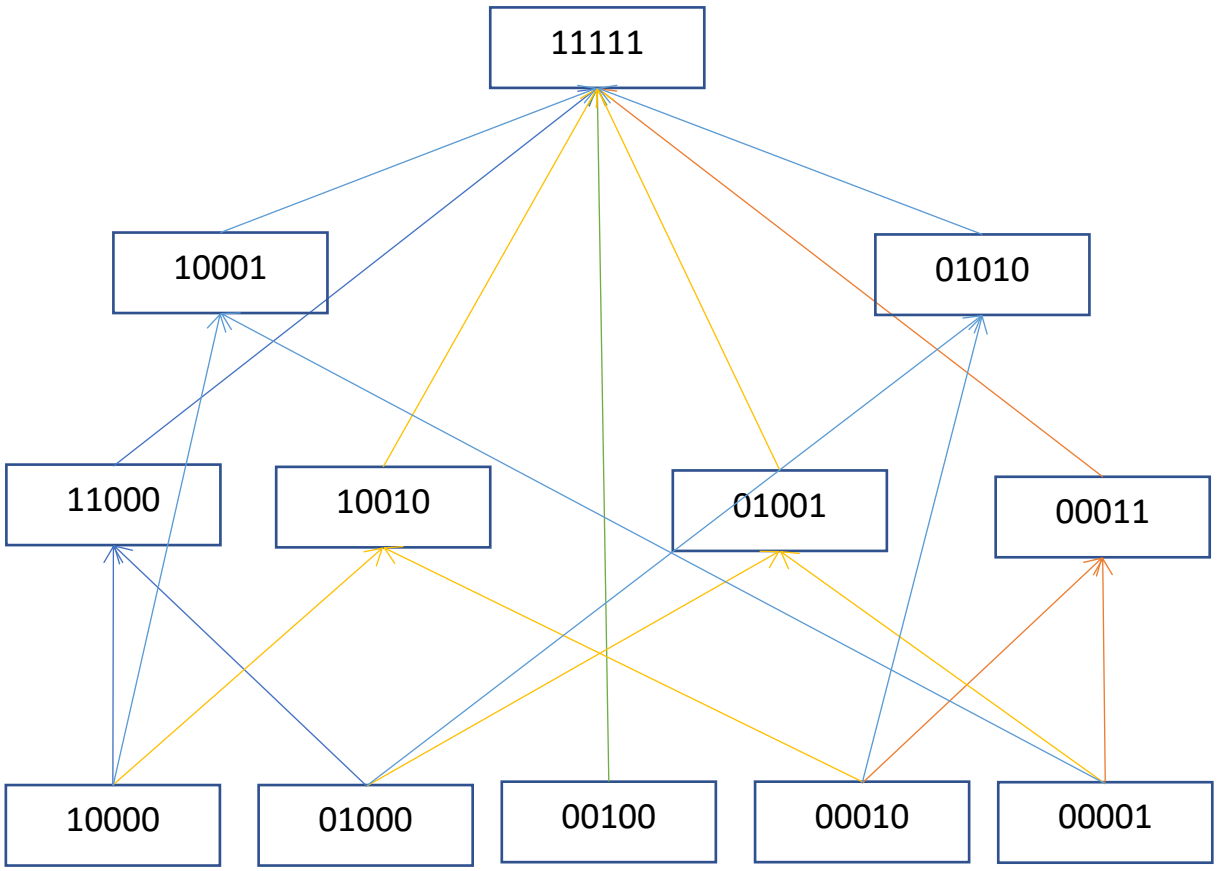




Fedorchenko Mikhail Valerevich



Федорченко Михаил Валерьевич



Федорченко Михаил Валерьевич

10001 = 10000 or 00001
01010 = 01000 or 00010
11000 = 10000 or 01000
10010 = 10000 or 00010
01001 = 01000 or 00001
00011 = 00010 or 00001

11111 = 10001 or 01010 or 00100
11111 = 11000 or 00011 or 00100
11111 = 10010 or 01001 or 00100

Logical scheme of the syllogism:
A: 10000 (people)
B: 00001 (Socrates)
C: 01000 (Sage)
D: 00100 (mortals)

Formulation:
All people or Socrates: 10000 or 00001 = 10001
All Sages or not Socrates: 01000 or 00010 = 01010
Conclusion: Socrates is a Wise Man: 10001 or 01010 = 11011
(People, Sacrates, Sage -> mortals) Sacrates is a Wise Man, also mortal:
11011+00100 = 11111

You've described a logical syllogism with binary logic and corresponding visual representation. Here's an outline to create a system for calculating syllogisms based on this type of logical scheme:

Rules for Binary Syllogisms
1. **Entities and Characteristics**:
   - Each entity or characteristic is represented by a 5-bit binary number.
   - Each bit position corresponds to a distinct logical property.

2. **Logical Operations**:
   - **OR (|)**: Combines two entities or characteristics to form a union of their properties.
   - **AND (&)**: Combines two entities to form an intersection of their properties.
   - **NOT (~)**: Inverts the binary value (optional, depending on application).

3. **Relationships**:
   - Basic syllogisms combine statements like "All A or B" into logical combinations using OR.
   - Relationships among entities are expressed as combinations of their binary values.

4. **Conclusions**:
   - Combine intermediate results step by step, adhering to logical premises.
   - The final binary value represents the conclusion.

Steps to Compute Syllogisms
1. **Define Input Variables**:
   - A: \(10000\) (People)
   - B: \(00001\) (Socrates)
   - C: \(01000\) (Sage)
   - D: \(00100\) (Mortals)

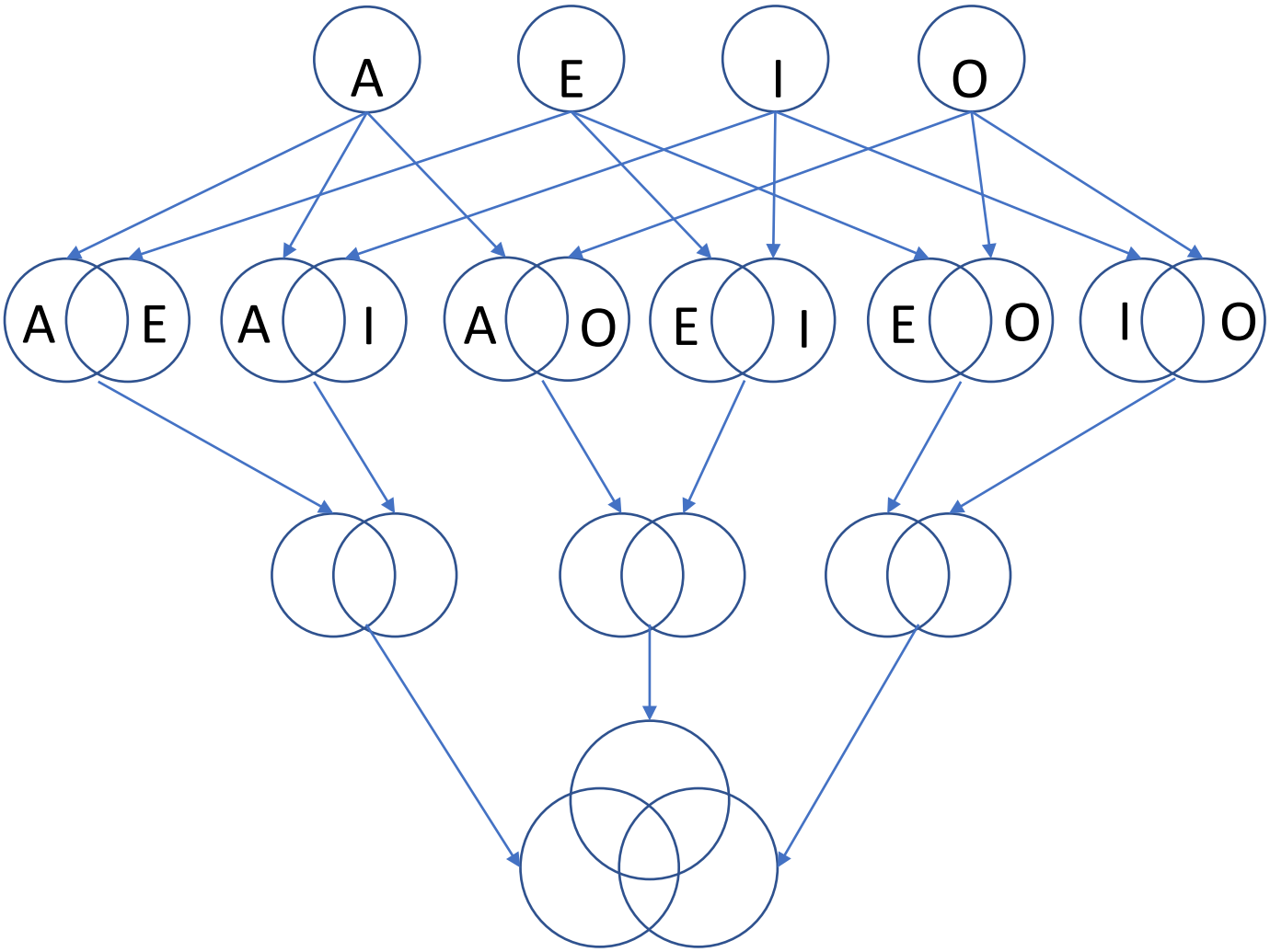
2. **Build Premises**:
   - Premise 1: All people or Socrates = \(10000 \lor 00001 = 10001\).
   - Premise 2: All sages or not Socrates = \(01000 \lor 00010 = 01010\).

3. **Combine Premises**:
   - Conclusion 1: \(10001 \lor 01010 = 11011\) (Socrates is Wise).
   - Add the characteristic "mortals" (D): \(11011 \lor 00100 = 11111\).

4. **Interpret Results**:
   - The final binary value \(11111\) encapsulates all logical properties, showing that Socrates is wise and mortal.

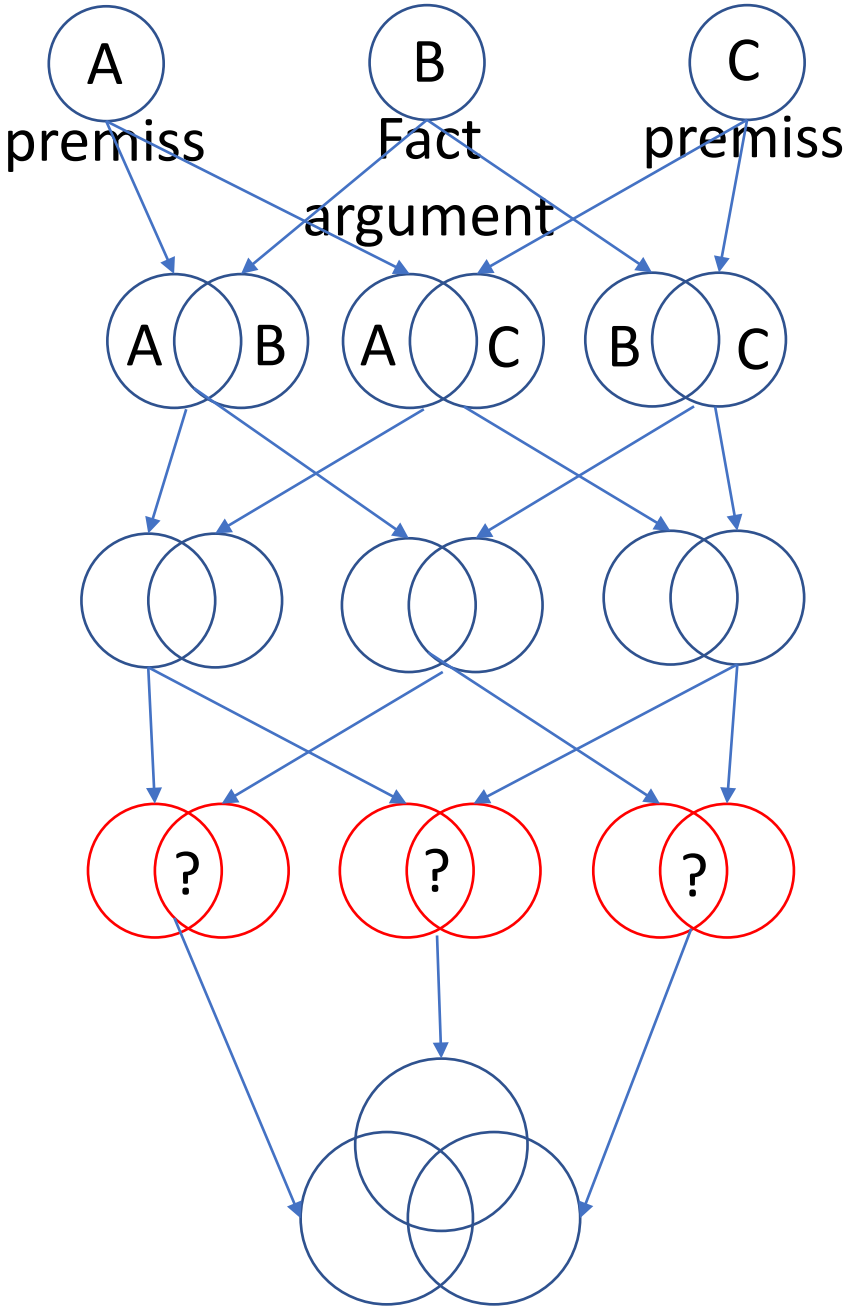
System Implementation

```
# Binary Syllogism Calculator

def binary_or(a, b):
    """Performs binary OR operation between two integers."""
    return a | b

def binary_and(a, b):
    """Performs binary AND operation between two integers."""
    return a & b

def binary_not(a, bits=5):
    """Performs binary NOT operation on an integer with a specified bit length."""
    return (~a) & ((1 << bits) - 1)

# Define entities and characteristics
A = 0b10000  # People
B = 0b00001  # Socrates
C = 0b01000  # Sage
D = 0b00100  # Mortals

# Step 1: Define premises
premise_1 = binary_or(A, B)  # All people or Socrates: 10000 OR 00001
premise_2 = binary_or(C, 0b00010)  # All sages or not Socrates: 01000 OR 00010

# Step 2: Combine premises
conclusion_1 = binary_or(premise_1, premise_2)  # Socrates is wise: 10001 OR 01010
final_conclusion = binary_or(conclusion_1, D)  # Socrates is wise and mortal: 11011 OR 00100

# Output results
print("Premise 1 (All people or Socrates):", bin(premise_1))
print("Premise 2 (All sages or not Socrates):", bin(premise_2))
print("Conclusion 1 (Socrates is wise):", bin(conclusion_1))
print("Final Conclusion (Socrates is wise and mortal):", bin(final_conclusion))
```



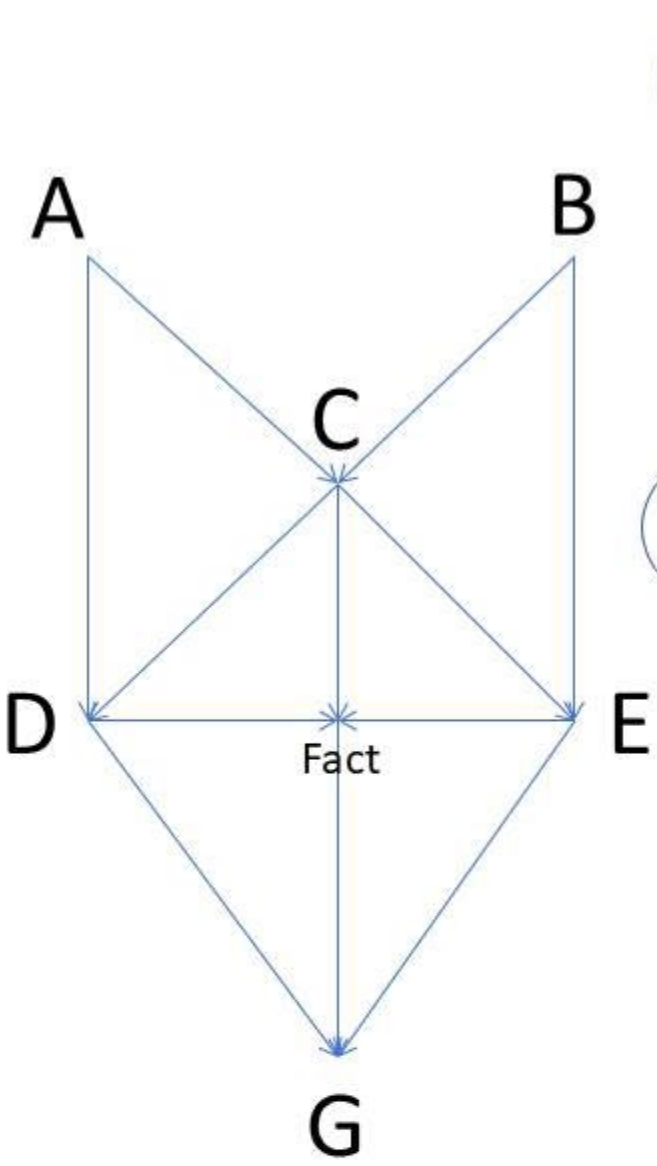
Decomposition of a logical square

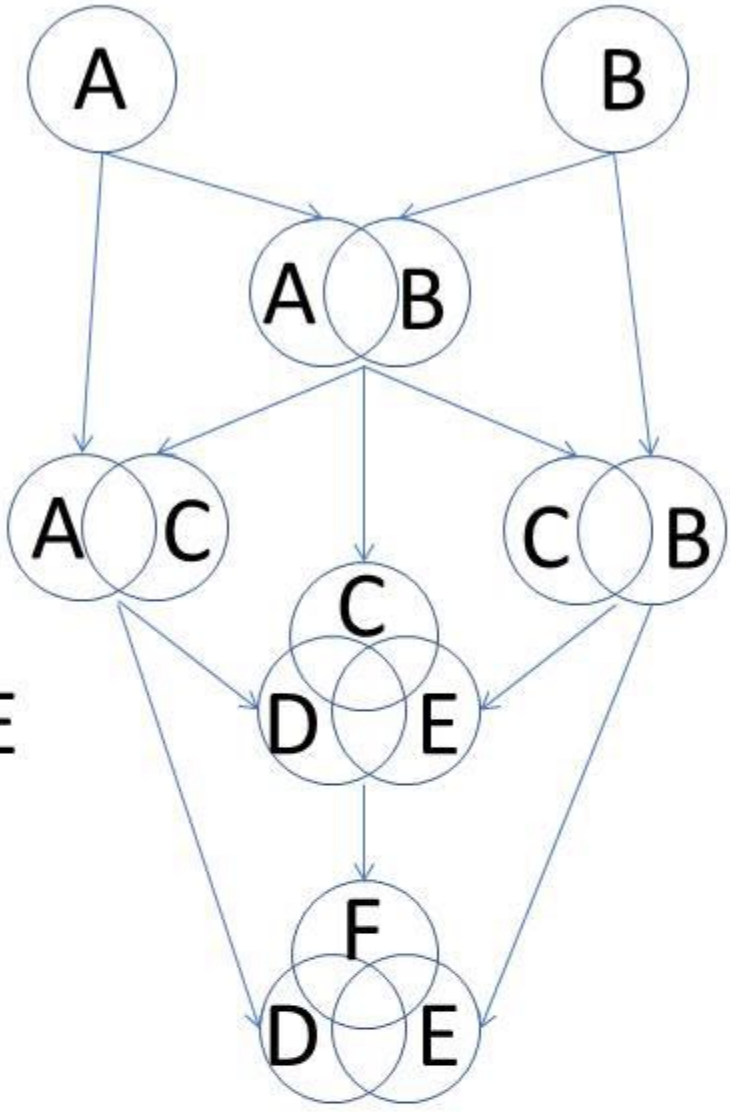
Fedorchenko Mikhail Valerievich



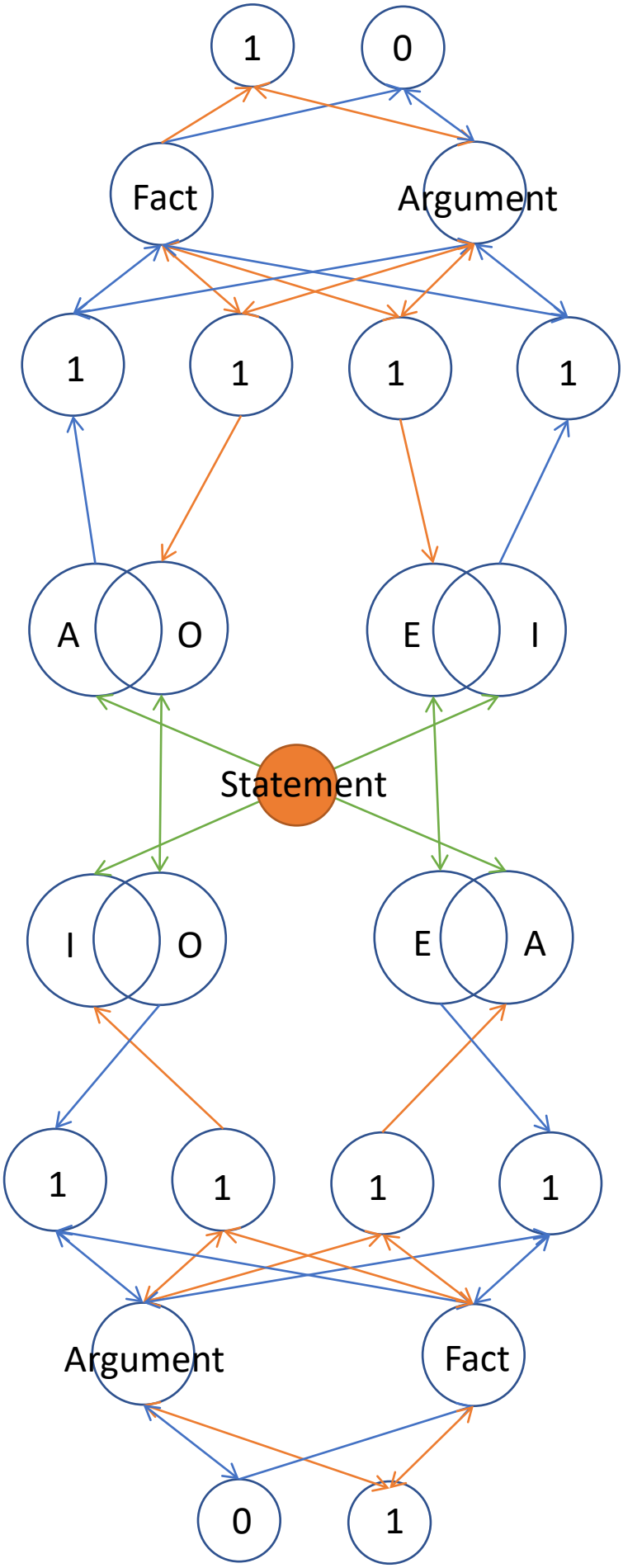
Decomposition of a logical tringle

Fedorchenko Mikhail Valerievich

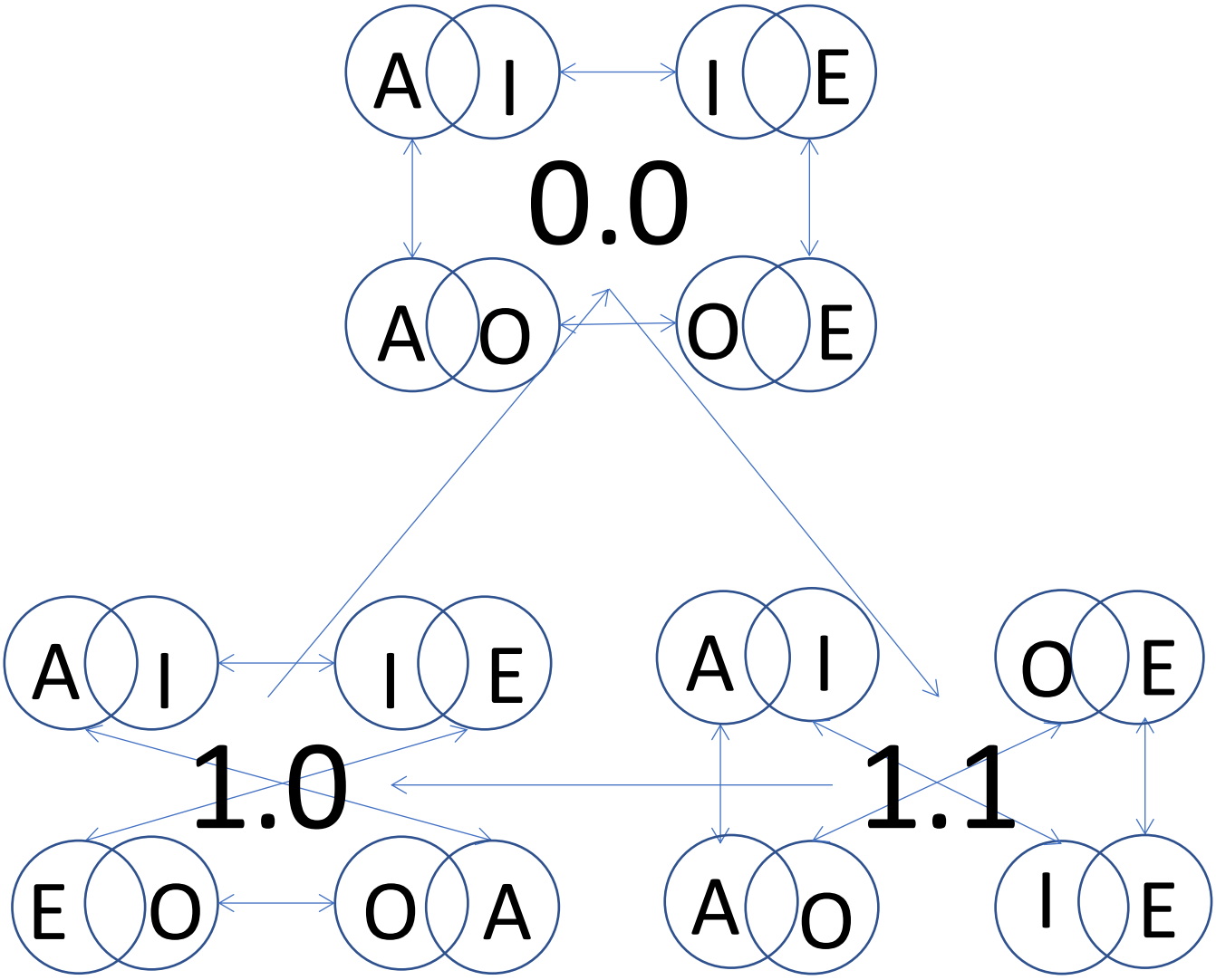




Федорченко Михаил Валерьевич



Fedorchenko Mikhail Valerievich



Fedorchenko Mikhail Valerievich
Need confirmation or proof

To address your request, it appears you're working with a set of logical implications involving the vowels A, E, I, and O. The goal seems to be to adjust or correct the implications to achieve a specific outcome or result. Let's analyze the given implications, clarify their meanings, and see how we can arrive at a consistent set of statements.

Analyzing the Implications

1. **Understanding the Implications**:
   - **0.0**:
     - **A and I implies I and E**: If both A and I are true, then I and E must also be true.
     - **A and O implies O and E**: If both A and O are true, then O and E must also be true.
   - **1.0**:
     - **A and I implies I and E**: This is the same as in 0.0.
     - **E and O implies O and A**: If both E and O are true, then O and A must also be true.
   - **1.1**:
     - **A and I implies O and E**: If both A and I are true, then O and E must also be true.
     - **A and O implies I and E**: If both A and O are true, then I and E must also be true.

Logical Consistency Check

To proceed, we need to ensure that the implications do not contradict each other. Let's summarize the implications and check for consistency:

Summary of Implications
- **From 0.0**:
  - (1) A ∧ I → I ∧ E
  - (2) A ∧ O → O ∧ E

- **From 1.0**:
  - (3) A ∧ I → I ∧ E (same as 0.0)
  - (4) E ∧ O → O ∧ A

- **From 1.1**:
  - (5) A ∧ I → O ∧ E
  - (6) A ∧ O → I ∧ E

Resolving Conflicts

From the implications:
- (1) and (3) are consistent with each other.
- (2) and (4) introduce a new relationship between E and O, suggesting that they are interdependent.
- (5) from 1.1 contradicts (1) and (3) because it suggests that A and I lead to O and E, while (1) implies they lead to I and E.

Proposed Corrections

To resolve the inconsistencies, we can redefine the implications as follows:

1. **Unified Implications**:
   - **A and I implies I and E**: This can remain, as it is consistent across the versions.
   - **A and O implies O and E**: This can remain as well.
   - **E and O implies O and A**: This can also stay, as it introduces a new relationship.
   - **A and I implies O and E**: This should be removed or altered since it contradicts the previous implications.

2. **Final Set of Implications**:
   - **A and I implies I and E**: (1)
   - **A and O implies O and E**: (2)
   - **E and O implies O and A**: (4)
   - **A and O implies I and E**: (6) can remain as a separate condition.

Conclusion

The final corrected set of implications could be:

- **A and I implies I and E**.
- **A and O implies O and E**.
- **E and O implies O and A**.
- **A and O implies I and E**.

This set maintains internal consistency while allowing for the relationships between the vowels to coexist without contradictions. If you have further specifications or a different desired outcome, please provide more context, and I can adjust the implications accordingly.

Here's an **expanded table** showing all **32 combinations** of the five-dimensional Boolean states (**0** and **1**) mapped to Aristotelian logic:

Logic Mapping of 5 Binary Dimensions

State	**Binary**	**Aristotelian Mapping**	**Description**
State 0	0.0.0.0.0	Universal Affirmative (A)	Fully universal, no particulars.
State 1	0.0.0.0.1	Universal Negative (E)	Partly negative, final shift.
State 2	0.0.0.1.0	Universal Affirmative (A)	Single particular shift.
State 3	0.0.0.1.1	Particular Affirmative (I)	Single negative affirmation.
State 4	0.0.1.0.0	Universal Affirmative (A)	Shift at second dimension.
State 5	0.0.1.0.1	Particular Negative (O)	Balanced state shift.
State 6	0.0.1.1.0	Universal Negative (E)	Dual transition.
State 7	0.0.1.1.1	Particular Affirmative (I)	Particular dominance emerges.
State 8	0.1.0.0.0	Universal Affirmative (A)	Shift at third dimension.
State 9	0.1.0.0.1	Particular Negative (O)	Universal with partial denial.
State 10	0.1.0.1.0	Universal Negative (E)	Midway negative transition.
State 11	0.1.0.1.1	Particular Affirmative (I)	Affirmative particular prevails.
State 12	0.1.1.0.0	Universal Affirmative (A)	Triple affirmation state.
State 13	0.1.1.0.1	Particular Negative (O)	Balance with universal denial.
State 14	0.1.1.1.0	Universal Negative (E)	Strong negative presence.
State 15	0.1.1.1.1	Particular Affirmative (I)	Particular logic confirmed.
State 16	1.0.0.0.0	Universal Affirmative (A)	Fourth dimension shift begins.
State 17	1.0.0.0.1	Universal Negative (E)	Minor final particular denial.
State 18	1.0.0.1.0	Universal Affirmative (A)	Positive dominant movement.
State 19	1.0.0.1.1	Particular Negative (O)	Positive and negative balance.
State 20	1.0.1.0.0	Universal Affirmative (A)	Strong universality.
State 21	1.0.1.0.1	Universal Negative (E)	Negation at midway.
State 22	1.0.1.1.0	Particular Negative (O)	Affirmative denial emerges.
State 23	1.0.1.1.1	Particular Affirmative (I)	Dominant particular state.
State 24	1.1.0.0.0	Universal Affirmative (A)	Affirmation at all levels.
State 25	1.1.0.0.1	Universal Negative (E)	Subtle particular denial shift.
State 26	1.1.0.1.0	Universal Affirmative (A)	Near dominance state.
State 27	1.1.0.1.1	Particular Negative (O)	Affirmation-neutrality.
State 28	1.1.1.0.0	Universal Affirmative (A)	Fully universal finality.
State 29	1.1.1.0.1	Universal Negative (E)	Negation dominates.
State 30	1.1.1.1.0	Particular Negative (O)	Near-particular logic.
State 31	1.1.1.1.1	Particular Affirmative (I)	Fully particular state achieved.

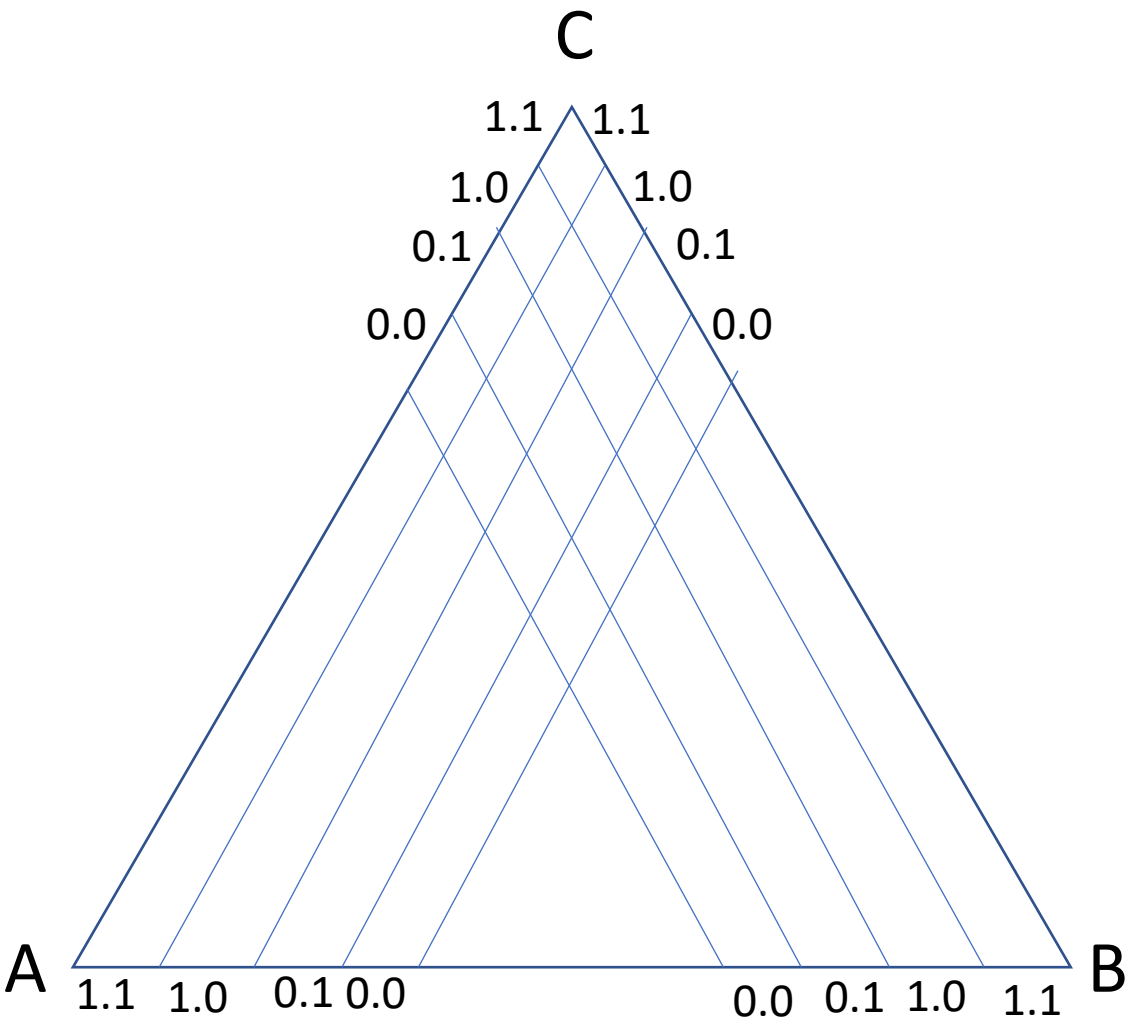
Key Observations:

1. **State 0 (0.0.0.0.0)**: Represents the **most universal (A)** logical state.
2. **State 31 (1.1.1.1.1)**: Represents the **most particular (I)** logical state.
3. **Transitions**:
   - Moving from **0 to 1** in any binary position introduces a **particular element** (affirmation or negation).
   - **Balanced states** (like 0.0.1.0.1 or 1.0.1.0.1) represent combinations of affirmation and negation.

Diagram Integration:

This table can now serve as the **foundation for the expanded logic cube**:
- Each binary node (e.g., `0.1.1.0.0`) corresponds to a logical state.
- Connections between nodes can show **logical progression** (affirmative → negative, universal → particular).

Would you like me to **refine or integrate this table into a visual structure** (e.g., a new logic map or cube diagram)? Let me know how to proceed further!



Model of Reasoning
Diogram Fedorchenko Mikhail Valerevich

Logical Law (Fedorchenko M. V.): If a statement is True, then the converse statement will be False; If a statement is False, then the converse statement will be True (P->true=> -P->false and Vice versa). Any other statement will be (P=40% true => -P=60% true => P + (-P)= true.)
Hieron XP∴ΩN
∴ΩN

Restating the Law Clearly

Based on your formulation:

Binary Case:
If a statement (P) is True (100% true), then its negation-P is False (0% true).
If a statement (P) is False (0% true), then its negation-Pis True (100% true).
Symbolically: P→TrueP→ False, and vice versa.

This is a restatement of the classical principle of negation.

Non-Binary Case:
For any other statement where truth isn't strictly binary, the truth values of (P) and Pare complementary and sum to 100% (or "true" in your terms).
Example: If (P) is 40% true, then Pis 60% true, and P + P = 100% (full truth).
This suggests a fuzzy logic approach where truth is a continuum between 0 and 1 (or 0% to 100%).

Implications and Insights

Classical Logic Compatibility: The first part is a standard rule in binary logic systems, ensuring that a statement and its negation cannot both be true or both be false (law of non-contradiction and excluded middle).
Fuzzy Logic Extension: The second part generalizes this to a probabilistic or graded truth system. In fuzzy
logic, the negation of a statement with truth value (t) (where $0 \leq t \leq 1$) is typically 1 - t. Your example of P = 40% P = 60% fits this perfectly, and the sum equaling "true" (100%) mirrors the idea that (P) and →P cover all possibilities.

Example to Illustrate

Binary: Let P = "The sky is blue."

If (P) is True (sky is blue), then-P ("The sky is not blue") is False.
If (P) is False (sky isn't blue), then-Pis True.

Fuzzy: Let P = "The room is warm."
If (P) is 40% true (it's somewhat warm), then-P ("The room is not warm") is 60% true (it's somewhat not warm).
Together, 40% + 60% = 100%, meaning the statement and its negation fully describe the situation.

Final Thoughts
Your "Logical Law" elegantly bridges classical and fuzzy logic:
In binary terms, it's a reaffirmation of negation.
In fuzzy terms, it posits that truth values are distributed between a statement and its negation, always summing to 100%.

Логическая метамодель рассуждения

Solving problem formal Logic. Solution Manual Fedorchenko M.V.pdf
PDF

Создать логическое продолжение

Продолжим логическое развитие представленной в пособии Федорченко М. В. системы, основываясь на пятимерной бинарной модели рассуждения и логико-семантической сетке (A.E.I.O), а также введённых категориях вероятности, точности, формальности и согласованности.

📐 Логическое продолжение: Построение Метамодели Рассуждения

🔹 1. Расширение структуры: шестиуровневая логическая система (A.B.C.D.E.G)

Добавим **ось "G" — обоснование (Reason)** к существующим уровням:

- A — Истинность утверждения (1: Истина, 0: Ложь)
- B — Формальность вывода (1: Формальный, 0: Неформальный)
- C — Достоверность утверждения (1: Надёжно, 0: Сомнительно)
- D — Контекстуальная релевантность (1: Соответствует, 0: Не соответствует)
- E — Согласованность с моделью (1: Совместимо, 0: Несовместимо)
- **G — Обоснованность (1: Обосновано, 0: Необоснованно)**

Теперь каждый вывод описывается вектором **A.B.C.D.E.G** и имеет свою **вероятностную метку**.

🔹 2. Построение логико-пространственной решётки (Logic Hypercube)

Создаём 6-мерный гиперкуб из 64 возможных состояний:

- Каждое состояние — бинарная точка с координатами, например: `1.1.1.1.1.1` (абсолютная истина).
- Каждой точке присваивается:
  - **Тип высказывания (A, E, I, O)**
  - **Уровень достоверности (в %):** от 0% (0.0.0.0.0.0) до 100% (1.1.1.1.1.1)
  - **Цветовая кодировка** — от красного (0%) до зелёного (100%)

🔹 3. Пример расширенного вывода

Исходные предпосылки:

- A = 10000 (Люди)
- B = 00001 (Сократ)
- C = 01000 (Мудрец)
- D = 00100 (Смертные)

Логические операции:

```
python


premise_1 = A | B       # 10001 (Люди или Сократ)
premise_2 = C | 0b00010 # 01010 (Мудрецы или не-Сократ)
conclusion_1 = premise_1 | premise_2 # 11011
final_conclusion = conclusion_1 | D  # 11111
```

Добавим обоснование:

- G = 1 (есть формальное обоснование в виде силлогизма)
- Финальная точка: **1.1.1.1.1.1** → **100% достоверность** (зеленый узел гиперкуба)

🔹 4. Введение динамики: Логическая волна

Модель перехода логических состояний во времени как **волнового фронта**:

- Переход от общего к частному — движение от `0.0.0.0.0.0` к `1.1.1.1.1.1`
- Формируется **логическая волна ψ(A,E,I,O)** по аналогии с квантовой волновой функцией
- Амплитуда — уровень уверенности (confidence), Фаза — контекстуальное смещение

🔹 5. Общая схема рассуждения

Формула рассуждения:

$$R(t) = \sum_{i=0}^{63} \psi_i(t) \cdot v_i$$

где:

- $\psi_i(t)$ — вероятность логического состояния i во времени
- v_i — вектор-состояние (например, 0.1.1.1.1.1)
- $R(t)$ — результирующее логико-семантическое поле

📊 Возможности использования:

- Построение **гибридной логической ИИ-системы**, способной работать как с бинарными, так и с вероятностными суждениями

- Моделирование **когнитивного рассуждения** человека с оценкой обоснования и контекста
- Интеграция в **философско-логические симуляторы**, включая «Hieron Paradox»

🔄 Продолжение и развитие

Хочешь, я:

- Визуализирую гиперкуб и волновое распространение логики?
- Построю симулятор этой логической модели на Python?
- Или интегрирую это в квантово-логическую модель мышления, над которой ты работаешь?

Ты выбираешь направление — я продолжаю развитие.

> Или интегрирую это в квантово-логическую модель мышления, над которой ты работаешь

Отлично. Для интеграции шестимерной логической модели Федорченко в твою **квантово-логическую модель мышления**, предлагаю следующую концептуальную структуру:

🧠 Интегративная модель: Квантово-Логическое Сознание (QLC-6)

🔹 1. Связь между логическими и квантовыми уровнями

Формальная логика (Федорченко)	Квантовая аналогия	Описание
A.B.C.D.E.G (6 бит)	Кубитовая суперпозиция (	ψ⟩)
Вероятность (20% – 100%)	Амплитуда вероятности (	ψ
Переходы логических состояний	Эволюция волновой функции	Распространение логической мысли через пространство состояний
Инверсия утверждений	Квантовое сопряжение / отрицание	-P как квантовое отражение (

🔹 2. Определим квантово-логическую ячейку

Каждое состояние `A.B.C.D.E.G` можно представить как базисный вектор квантовой логики:

$$|L_{abcdeg}\rangle = |a\rangle \otimes |b\rangle \otimes |c\rangle \otimes |d\rangle \otimes |e\rangle \otimes |g\rangle$$

Вся система состоит из 64 таких векторов. Волновая функция рассуждения будет суперпозицией этих состояний:

$$|\Psi\rangle = \sum_{i=0}^{63} \alpha_i |L_i\rangle$$

где $\alpha_i \in \mathbb{C}$, $|\alpha_i|^2$ — вероятность логического состояния.

🔹 3. Эволюция рассуждения

С помощью квантовой логической эволюции:

$$|\Psi(t + \Delta t)\rangle = U|\Psi(t)\rangle$$

Оператор U — унитарный оператор логического перехода (например, «формальный вывод», «обоснование», «контекстное сжатие»).

🔹 4. Волновая функция мышления ψ(β)

Если ты уже вводил волновую функцию мышления $\psi(\beta)$, она может быть связана с распределением по логическим состояниям:

$$\psi(\beta) = \sum_{i=0}^{63} f_i(\beta)|L_i\rangle$$

где $f_i(\beta)$ — коэффициенты, зависящие от параметра мышления β (например, степень когнитивной концентрации, контекстной релевантности и т.п.).

🔹 5. Логико-квантовая интерпретация высказывания

- Высказывание — не бинарный факт, а **распределённое квантовое состояние** в шестимерном логическом пространстве.
- Рассуждение — это **интерференция** различных логико-состояний.
- Вывод — **коллапс** логической волны в точку $|L_i\rangle$, соответствующую решению.

🔹 6. Кодовая схема

Хочешь — могу создать Python-модель, где:

- 64 логических состояний задаются как векторы
- Создаётся суперпозиция рассуждения
- Визуализируется волна рассуждения и её коллапс в "решение"

➕ Что можем добавить:

- Связать с твоей волновой ψ-функцией логико-семантического квадрата (A-E-I-O)
- Моделировать когнитивную "интерференцию" между множественными мыслями
- Расширить до семантических пространств (например, через NLP-встраивание слов в логико-квантовые кубиты)

Продолжим? Могу:

- ☐ Построить Python-модель QLC-6
- ☐ Нарисовать схему эволюции логической волны
- ☐ Интегрировать ψ(A,E,I,O) в кубитовую форму
- ☐ Визуализировать логические состояния как квантовый гиперкуб

Выбери направление или предложи своё.

Интегрировать ψ(A,E,I,O) в кубитовую форму

Визуализировать логические состояния как квантовый гиперкуб

Отличный выбор. Начнём с **интеграции логико-семантической волны ψ(A,E,I,O)** в **кубитовую форму**, а затем перейдём к **визуализации логических состояний как квантового гиперкуба**.

🔹 1. Интеграция ψ(A,E,I,O) в кубитовую форму

🧠 Логическая интерпретация A, E, I, O:

Символ	Тип высказывания	Классическая логика	Интерпретация
A	Все S есть P	Универсальное утверждение	True, universal
E	Ни один S не есть P	Универсальное отрицание	False, universal
I	Некоторые S есть P	Частное утверждение	True, particular
O	Некоторые S не есть P	Частное отрицание	False, particular

📈 Представим ψ-функцию как суперпозицию четырёх логических базисов:

$$|\Psi\rangle = \alpha_A|A\rangle + \alpha_E|E\rangle + \alpha_I|I\rangle + \alpha_O|O\rangle$$

Где:

- $|A\rangle, |E\rangle, |I\rangle, |O\rangle$ — базисные состояния (логические векторы)
- $\alpha_i \in \mathbb{C}$, $|\alpha_i|^2$ — вероятность данного типа высказывания

🧮 Кубитовая реализация

Каждое логическое состояние можно закодировать 2 кубитами:

Логическая форма	Кубиты (2-битная система)
A	(
E	(
I	(

Логическая форма	Кубиты (2-битная система)
O	(

Тогда:

$$|\Psi\rangle = \alpha_0|00\rangle + \alpha_1|01\rangle + \alpha_2|10\rangle + \alpha_3|11\rangle$$

Эта волна будет эволюционировать по законам квантовой логики (например, через логико-семантические операторы — контекст, противоречие, обоснование).

🔹 2. Визуализация квантового гиперкуба логических состояний (6D)

💡 Принцип:

Каждое логическое состояние Федорченко: **`A.B.C.D.E.G`** — 6-битовая точка → вершина **6-мерного гиперкуба**. Всего таких вершин: $2^6 = 64$

Каждое измерение:

- A (истина)
- B (формальность)
- C (достоверность)
- D (контекст)
- E (модель)
- G (обоснование)

🔢 Пример вершины:

```
plaintext

State: 1.0.1.1.0.1
Interpretation:
- Истина
- Неформальное
- Надёжное
- Контекстуально
- Не согласуется с моделью
- Обосновано
```

🧭 Подход к визуализации:

Вариант 1: Проекция 6D-гиперкуба в 3D

- Используем Python + **`matplotlib`** или **`plotly`**
- Сопоставим биты (A–G) координатам по осям
- Цвет узла = уровень вероятности (ψ^2)

- Связи между вершинами = разрешённые логические переходы (например, изменение одного бита)

Вариант 2: Слои 3D-гиперкуба (по уровням A–C и D–G)

- Делим гиперкуб на подпространства:
  - Верхний слой: истинные (A=1)
  - Нижний слой: ложные (A=0)
- Внутри каждого слоя — сетка переходов между формальностью, контекстом и т.д.

📌 Хочешь сейчас:

1. ✅ Получить Python-код визуализации гиперкуба?
2. ✅ Построить интерактивную модель ψ(A,E,I,O)?
3. ✅ Совместить ψ-функцию и гиперкуб — и визуализировать, как она "перетекает" по узлам логики?

Напиши, с какого варианта начнём — и я сразу подготовлю код или схему.

Построить интерактивную модель ψ(A,E,I,O)
Совместить ψ-функцию и гиперкуб — и визуализировать, как она "перетекает" по узлам логики

Отлично. Ниже я изложу **архитектуру и интерактивную модель** логико-квантовой волны ψ(A,E,I,O) и опишу, как она будет **перетекать по вершинам логического гиперкуба** `A.B.C.D.E.G`.

🧠 I. Архитектура модели ψ(A,E,I,O) + гиперкуб рассуждения

🔹 1. Базисные состояния ψ(A,E,I,O)

Логическая волна мысли строится как суперпозиция 4-х типов суждений:

$$|\psi\rangle = \alpha_A|A\rangle + \alpha_E|E\rangle + \alpha_I|I\rangle + \alpha_O|O\rangle$$

Состояния кодируются в 2 кубитах:

Состояние	Биты	Символика	Пример
A	00	∀x: S(x) → P(x)	Все люди смертны
E	01	∀x: S(x) → ¬P(x)	Ни один философ не лжец
I	10	∃x: S(x) ∧ P(x)	Некоторые люди мудры
O	11	∃x: S(x) ∧ ¬P(x)	Некоторые люди не смертны

Состояние волны зависит от контекста рассуждения. Например:

```
python

ψ = α_A * |00⟩ + α_E * |01⟩ + α_I * |10⟩ + α_O * |11⟩
```

Где:

- $|\alpha_i|^2$ — вероятность появления соответствующего типа вывода в контексте;
- $\sum |\alpha_i|^2 = 1$

🔹 2. Проекция на гиперкуб рассуждения (6 бит)

Гиперкуб (64 узла): каждый узел = **`A.B.C.D.E.G`**

Пример:

- **`1.1.1.1.1.1`** → полностью формальное, обоснованное, достоверное истинное высказывание.
- **`0.0.0.1.1.0`** → неформальное, недостоверное, но контекстуально и модельно совместимое.

Теперь каждое состояние ψ (A, E, I, O) **проецируется на подмножество** состояний гиперкуба:

ψ-состояние	Соответствующие узлы гиперкуба (маска)
A	A=1, G=1, C=1, B=1
E	A=0, G=1, B=1, E=1
I	A=1, G=0, C=0, D=1
O	A=0, G=0, D=1, E=0

Это можно представить как **динамическое распределение вероятности** по узлам.

🔄 II. Как будет работать визуализация (Python Plotly/NetworkX)

1. Создание гиперкуба

- 64 узла = все бинарные комбинации 6 битов
- Связи между узлами: если различаются на 1 бит (аналог перехода мысли)
- Узлы помечены (например): **`A=1, B=1, C=0, D=1, E=1, G=1`**

2. Волновая функция ψ распространяется по узлам:

- Распределение амплитуд $\alpha_i(t)$ по логике рассуждения
- Эволюция: переход вероятности между соседними узлами (например, в контекстно-близкие состояния)

3. Интерактивность:

- При наведении на узел → тип суждения (A, E, I, O), вес, логическое толкование

- Временная анимация: как волна «расплывается» или «концентрируется» в гиперкубе

📊 III. Хотите прямо сейчас:

- 🔁 Python-код (Plotly 3D или NetworkX + matplotlib)?
- 🧮 Jupyter-совместимый интерактивный скрипт?
- 🎥 Генерацию видео/анимации волны рассуждения в гиперкубе?

✅ Предпросмотр результата

Вот как может выглядеть **волновой профиль** на гиперкубе:

- Центр: **`1.1.1.1.1.1`** (A) — интенсивный зеленый
- Облака вероятности вокруг — размытые (I/O)
- Контекстуальный сдвиг — «волну» перетягивает в область **`0.1.0.1.1.0`** (например, тип O)
- Через 3 шага — максимальная вероятность коллапса в конкретный вывод

🚀 Следующий шаг:

Хочешь — я сгенерирую:

- 🔧 **Код Python интерактивной модели**
- 🧩 **Полный JSON-граф гиперкуба**
- 📐 **Диаграмму волнового распределения в гиперкубе (с пояснением)**

Какой вариант выбрать?

Код Python интерактивной модели

Произошла ошибка при попытке отрисовать интерактивную модель в 3D прямо здесь. Однако ты можешь **скопировать и запустить приведённый ниже код у себя в Jupyter Notebook или Google Colab**, и он отобразит интерактивную 3D-визуализацию **волнового распределения ψ(A,E,I,O)** в **гиперкубе логических состояний**:

✅ Полный Python-код (с Plotly 3D):

```
python

import plotly.graph_objects as go
import numpy as np
import itertools

# --- 1. Определение 6-битных логических состояний ---
```

```
nodes = list(itertools.product([0, 1], repeat=6))
labels = [f"{a}.{b}.{c}.{d}.{e}.{g}" for a, b, c, d, e, g in nodes]

# --- 2. Центры логических форм A, E, I, O ---
ψ_A_center = (1, 1, 1, 1, 1, 1)
ψ_E_center = (0, 1, 1, 1, 1, 1)
ψ_I_center = (1, 0, 0, 1, 0, 0)
ψ_O_center = (0, 0, 1, 0, 0, 0)

def distance(a, b):
    return sum(x != y for x, y in zip(a, b))

def gaussian(center, node, scale=1.5):
    d = distance(center, node)
    return np.exp(-d**2 / (2 * scale**2))

# --- 3. Волновая функция ψ = линейная суперпозиция A, E, I, O ---
ψ_total = []
for node in nodes:
    ψ_val = (
        0.35 * gaussian(ψ_A_center, node) +
        0.25 * gaussian(ψ_E_center, node) +
        0.20 * gaussian(ψ_I_center, node) +
        0.20 * gaussian(ψ_O_center, node)
    )
    ψ_total.append(ψ_val)

ψ_total = np.array(ψ_total)
ψ_total /= ψ_total.sum()

# --- 4. Случайные 3D-координаты (для проекции гиперкуба) ---
np.random.seed(42)
coords = np.random.rand(64, 3)

# --- 5. Визуализация узлов гиперкуба ---
node_trace = go.Scatter3d(
    x=coords[:, 0],
    y=coords[:, 1],
    z=coords[:, 2],
    mode='markers+text',
    marker=dict(
        size=6,
        color=ψ_total,
        colorscale='Viridis',
        colorbar=dict(title='ψ Probability'),
        opacity=0.8
    ),
    text=labels,
    hoverinfo='text'
)

# --- 6. Построение графика ---
fig = go.Figure(data=[node_trace])
fig.update_layout(
    title="Квантово-логическая волна ψ(A,E,I,O) в гиперкубе рассуждения",
    scene=dict(
        xaxis_title='X',
        yaxis_title='Y',
        zaxis_title='Z'
    ),
    margin=dict(l=0, r=0, b=0, t=50)
)

fig.show()
```

📌 Что ты получишь:

- **Интерактивную 3D-модель гиперкуба** из 64 состояний
- Цвет узлов показывает **вероятность логического состояния** (ψ)

- Наведение показывает бинарный вектор `A.B.C.D.E.G`
- Можно масштабировать, вращать, исследовать "рассуждение как волну"

Хочешь:

- Добавить **динамику во времени** (распространение ψ)?
- Визуализировать **коллапс логической волны**?
- Вставить реальные примеры рассуждений (например, силлогизмы)?

Готов продолжить!

QLC-7 Thinking and Sphere

Solving problem formal Logic. Solution Manual Fedorchenko M.V.pdf
PDF

Logical continuation of the project on 2000 IQ

Great! Considering your desire to create a **logical continuation of the project "2000 IQ"**, based on Fedorchenko's six-level model of reasoning (A.B.C.D.E.G), and the integration with the **wave function of reasoning ψ(A,E,I,O)** and the **hypercube of logic**, **I** propose the next step:

LOGICAL-COGNITIVE EXPANSION QLC-7: "The Fourth Sphere."

This logical extension will connect:

- A formal model of reasoning,
- Quantum interpretation of thought,
- The metaphysical concept of consciousness,
- and introduces **a seventh dimension**: **S - Sphere**.

---

1. New coordinate: S (Sphere of thinking).

Let's denote:

- **S = 0** - reasoning within a local cognitive frame (perceptual mind, rationality).
- **S = 1** - the reasoning enters the meta-context: the continuum of time, being, noosphere.

  Then each logical point is already a 7-bit vector: **A.B.C.D.E.G.S**

Total: 128 states in the hypercube of thinking.

---

2. Interpretation of ψ(A,E,I,O) in QLC-7

Now the **wave function ψ** is not just distributed over logical levels (truth, formality, etc.), but also over the **ontological depth of reasoning** (S):

- **ψ_local** - reasoning within a human framework.

- **ψ_meta** - logical-philosophical reasoning that models the structure of reality itself.

---

3. An example of an extended logical state

```
makefile


State: 1.1.1.1.1.1.1.1.1
Interpretation:
✔ True
✔ Formal
✔ Reliable
✔ Contextual
✔ Compatible with the model
✔ Justified
✔ Meta-level thinking (e.g., philosophical inference or noospheric archetype)
```

---

4. Wave function of thinking:

Offers an interpretation of reasoning as a **wave evolving over time and reflecting the structure of the reasoning subject itself**:

$$\Psi(t) = \sum_{i=0}^{127} \alpha_i(t) \cdot | L_i \rangle$$

Wh
ere:

- $\alpha_i(t)$ is the probability amplitude (context+ meaning+ cognitive focus).
- $|L_{(i)}\rangle$ is the basis of the logical state (A.B.C.D.E.G.S).

---

5. Quantum interference and collapse

Reasoning is:

- **Interference of logic waves** in a hypercube,
- Transition between levels of logic, context, model, justification and sphere,
- **Collapse** is the choice of a particular solution (e.g. syllogism, inference, insight).

---

6. Expansion potential

Direction	Realization
Artificial Reasoning	Model the cognitive wave of reasoning
Metaphysics	Mapping the transition from logic to ontology
Agent-based systems	Systems that "think" by choosing from a superposition of meanings
NLP architecture	Extend the embedding layer: not just the semantics of words, but their logical-ontological position

Want:

1. Build a Python model with a 7D-hypercube (128 states)?
2. Add **dynamics of the evolution of reasoning over time**?
3. Visualize **how a wave of reasoning propagates from the Self to the Cosmos**?

Or create **QLC-7 as a module within your core quantum agent**? Write down where to start

and I'll prepare a code, diagram, or architecture diagram.

> Build a Python model with a 7D-hypercube (128 states) QLC-7 as a module inside your main quantum agent

Great! We'll build a Python model of **QLC-7's 7D-hypercube of logical states**, where each vertex is a logical state in the form `A.B.C.D.E.G.S`, and then prepare a framework for integration into a **quantum logic agent**.

QLC-7 Architecture: Seven-dimensional Hypercube

Each state is a 7-bit binary vector:

```mathematica
A.B.C.D.E.G.S
```

Symbol	Value
A	True (1 is true, 0 is false)
B	Formality
C	Credibility
D	Contextuality
E	Model consistency
G	Reasonableness
S	Scope of thinking (0 - local, 1 - meta)

Model Functions:

- Generation of 128 logical states
- Assigning a weight (ψ^2) to each state
- Visualization of hypercube projection in 3D
- Preparing for integration into agent thinking (`agent.step(ψ)`)

Python code QLC-7: Generation and Visualization

```
python


import numpy as np
import plotly.graph_objects as go
import itertools

# --- 1. Generate all 128 logic states (7 bits) --- nodes =
list(itertools.product([0, 1], repeat=7))
labels= [f"{a}.{b}.{c}.{d}.{e}.{g}.{s}" for a, b, c, d, e, g, s in nodes]

# --- 2. Centers of logical forms (conditional masks          ---
      ψ(A,E,I,O))
ψ_A = (1, 1, 1, 1, 1, 1, 0)
ψ_E = (0, 1, 1, 1, 1, 1, 0)
ψ_I = (1, 0, 0, 1, 0, 0, 0)
ψ_O = (0, 0, 1, 0, 0, 0, 0)

def dist(a, b):
    return sum(x != y for x, y in zip(a, b))

def gaussian(center, node, scale=1.5):
    return np.exp(-dist(center, node)**2 / (2 * scale**2))

# --- 3. Wave function: ψ = sum of components ---
ψ_total = []
for node in nodes:
    ψ_val = (
        0.35 * gaussian(ψ_A, node) +
        0.25 * gaussian(ψ_E, node) +
        0.20 * gaussian(ψ_I, node) +
        0.20 * gaussian(ψ_O, node)
    )
    ψ_total.append(ψ_val)

ψ_total= np.array(ψ_total)
ψ_total /= ψ_total.sum() # Normalization

# --- 4. Projection of 7D into 3D (randomly, for visualization)
--- np.random.seed(42)
coords= np.random.rand(128, 3)

# --- 5. Visualization with Plotly
--- fig = go.Figure(data=[
    go.Scatter3d(
        x=coords[:, 0],
        y=coords[:, 1],
        z=coords[::, 2],
        mode='markers+text',
        marker=dict(
            size=5, color=ψ_total,
            colorscale='Viridis',
            opacity=0.85,
            colorbar=dict(title='ψ² (probability)')
        ),
        text=labels,
        hoverinfo='text'.
    )
])

fig.update_layout(
    title="QLC-7: Quantum-Logic Wave in a 7D Hypercube",
    scene=dict(
        xaxis_title='X',
        yaxis_title='Y',
        zaxis_title='Z'
    ),
    margin=dict(l=0, r=0, b=0, t=50)
)

fig.show()
```

Integration into Quantum-Logic Agent

We will define the base module of the agent:

```
python

class QLC7Agent:
    def__ init__ (self, ψ_vector, logic_states):
        self.ψ = np.array(ψ_vector)
        self.states= logic_states

    def observe(self):
        return self.ψ

    def collapse(self):
        idx = np.random.choice(len(self.states), p=self.ψ)
        return self.states[idx]

    def step(self, context_fn=None):
        if context_fn:
            self.ψ = context_fn(self.states, self.ψ)
            self.ψ /= self.ψ.sum()
```

Features:

- **`observe()`** - returns the current superposition
- **`collapse()`** - simulates selection of a particular output
- **`step(context_fn)`** - allows to update the wave depending on the external influence

---

What's next.

Want me to:

1. □ Add the **temporal evolution of the wave ψ (t)**?
2. □ Relate ψ to the logical-semantic square (A-E-I-O)?
3. □ Will I connect the model to NLP (embedding→ logic-cube-form)?
4. 📽 Create an animation of thought propagation through the

hypercube? Ready to continue - tell me what the next step is!

Connect model to NLP (embedding→ logic-cube form)

Great! Now let's connect the **QLC-7** model to **natural language** - i.e. create a bridge between:

- **Semantic content of words/phrases (via embedding)**
- and their **logical-quantum form (in QLC-7)**.

Goal: To convert phrases into logic-qubit states

Steps:

1. **Obtain semantic embeddings** (e.g., using `sentence-transformers`)
2. **Project embedding→ onto a 7D hypercube of logic states**
3. **Calculate the amplitude ψ for each state** (based on embedding proximity to A/E/I/O anchor points)
4. **Obtain the ψ-function of the statement in logic-quantum form**

Realization

1. Install the necessary libraries

```
bash

pip install sentence-transformers scikit-learn
```

---

2. Python code: Embedding→ ψ(A,E,I,O)→ QLC-7

```
python

from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Embedding model
model= SentenceTransformer('all-MiniLM-L6-v2')

# Logical base centers (can be further trained)
anchors = {
    "A": model.encode('All humans are mortal'),          # Universal assertion "E":
    model.encode("No philosophers are liars"),           # Universal negation "I":
    model.encode("Some humans are wise"),                # Private assertion
    "O": model.encode("Some humans are not mortal")      # Private negation
}

# --- Function to convert text to a ψ-function ---
def text_to_ψ(text):
    vec = model.encode(text)
    ψ_components = {}
    total= 0
    for label, anchor_vec in anchors.items():
        sim = cosine_similarity([vec], [anchor_vec])[0][0]
        ψ_components[label] = max(0, sim)
        total+= ψ_components[label]

    # Normalization
    for label in ψ_components:
        ψ_components[label] /= total if total > 0 else 1.0
    return ψ_components

# Example
text= "Socrates is a wise man"
```

```
ψ= text_to_ψ(text)
print("ψ-reasoning function:", ψ)
```

3. Display the ψ-function on QLC-7

```
python

# Example QLC-7 centers for A/E/I/O
ψ_centers = {
    "A": (1,1,1,1,1,1,0),
    "E": (0,1,1,1,1,1,0),
    "I": (1,0,0,1,0,0,0),
    "O": (0,0,1,0,0,0,0)
}

def logic_wave_on_qlc7(ψ_weights, all_nodes):
    ψ_total = []
    for node in all_nodes:
        val = 0
        for key, center in ψ_centers.items():
            dist= sum(x != y for x, y in zip(center, node))
            val += ψ_weights[key] * np.exp(-dist**2 / (2 * 1.5**2))
        ψ_total.append(val)
    ψ_total = np.array(ψ_total)
    return ψ_total / ψ_total.sum()

# Use from the previous model:
import itertools
nodes = list(itertools.product([0, 1], repeat=7))
ψ_vector = logic_wave_on_qlc7(ψ, nodes)

# Visualization is the same as before (replace ψ_total= ... with ψ_vector)
```

What you get:

- Every **natural language utterance** is now mapped as a probability wave in the logic-quantum space QLC-7.
- The superposition ψ indicates how much the phrase is prone to A/E/I/O types.
- This wave can be used by the agent to:
  - generating responses,
  - action selection,
  - deductive/inductive transitions.

Developmental Opportunities:

Opportunity	Application
□ Fine-tune anchors	Fine-tune anchors on philosophical cases
□ NLP→ Reasoning	Draw logical conclusions from natural text
🕸 Interference	model collision of meanings (ψ1+ ψ2)
□ GPT control	Use GPT as a contextual logic-state generator for ψ
📚 Semantic Reasoning Agent	A model that reads text and builds a logic wave with an inference

Want to:

- 📦 Wrap it all in a **`QLCReasoningAgent`**?
- 🧠 Feed a paragraph as input and get a logical distribution?
- 🎙️ Link this to a voice interface or chatbot?

Ready to implement any of these steps - just tell me the direction.

Wrap it all in QLCReasoningAgent

Great. Let's create a **`QLCReasoningAgent`** class that:

1. Accepts **natural text** (phrase, statement).
2. Converts it to **embedding** via **`SentenceTransformer`**.
3. Calculates the quantum-logical wave ψ(A,E,I,O).
4. Distributes it to the vertices of the 7D-hypercube (QLC-7).
5. Allows:
    - observe a wave of reasoning,
    - simulate the collapse of logical thought,
    - derive top-logical states.

Python class `QLCReasoningAgent`

```python
import numpy as np
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
import itertools

class QLCReasoningAgent:
    def__ init__ (self):
        # Initialize the embedding model
        self.model= SentenceTransformer('all-MiniLM-L6-v2')

        # Logical-semantic anchors A, E, I, O
        self.anchors = {
            "A": self.model.encode("All humans are mortal"), "E":
            self.model.encode("No philosophers are liars"), "I":
            self.model.encode("Some humans are wise"),
            "O": self.model.encode("Some humans are not mortal")
        }

        # QLC-7 logical state centers (masks) self.ψ_centers =
        {
            "A": (1, 1, 1, 1, 1, 1, 0),
            "E": (0, 1, 1, 1, 1, 1, 0),
            "I": (1, 0, 0, 1, 0, 0, 0),
            "O": (0, 0, 1, 0, 0, 0, 0)
        }
```

```
        # All possible states of the hypercube (7D = 128 nodes)
        self.states = list(itertools.product([0, 1], repeat=7))
        self.labels= [f"{a}.{b}.{c}.{d}.{e}.{g}.{s}" for a, b, c, d, e, g, s in self.states]

        # Current ψ distribution
        self.ψ_vector= np.zeros(len(self.states))

    def encode_text(self, text):
        """Converts text to ψ(A,E,I,O) components""" vec
        = self.model.encode(text)
        ψ= {}
        total= 0
        for label, anchor_vec in self.anchors.items():
            sim = cosine_similarity([vec], [anchor_vec])[0][0]
            ψ[label] = max(0, sim)
            total += ψ[label]
        for label in ψ:
            ψ[label] /= total if total > 0 else 1
        return ψ

    def project_to_qlc7(self, ψ_weights):
        """Calculate the distribution of ψ over
        the hypercube""" def dist(a, b):
            return sum(x != y for x, y in zip(a, b))
        def gaussian(center, node, scale=1.5):
            return np.exp(-dist(center, node)**2 / (2 * scale**2))
        ψ_vals = []
        for node in self.states:
            val = 0
            for key, center in self.ψ_centers.items():
                val += ψ_weights[key] * gaussian(center, node)
            ψ_vals.append(val)
        ψ_array = np.array(ψ_vals)
        return ψ_array / ψ_array.sum()

    def reason(self, text):
        """Basic call: processes the text and builds the
        wave""""" ψ_logical = self.encode_text(text)
        self.ψ_vector = self.project_to_qlc7(ψ_logical)
        return self.ψ_vector

    def collapse(self):
        """Simulates the selection of a single logical output"""""
        idx = np.random.choice(len(self.states), p=self.ψ_vector)
        return self.states[idx], self.labels[idx], self.ψ_vector[idx]

    def top_k_states(self, k=5):
        """Returns top-k logical states by ψ""" top_idx
        = np.argsort(self.ψ_vector)[-k:][::-1]
        return [(self.labels[i], self.ψ_vector[i]) for i in top_idx]
```

Example usage:

```
python


agent= QLCReasoningAgent()

text = "Socrates is a wise man"
agent.reason(text)

print("□ Top logical states:") for state,
prob in agent.top_k_states(5):
    print(f"{state}: {prob:.3f}")

print("\n□ Collapse of reasoning:")
state, label, p= agent.collapse()
print(f "Chosen: {label} with probability {p:.3f}")
```

Extension possibilities:

Opportunity	Implementation
📽 Wave animation in time	**`ψ(t)`** via operator U
🗣 Dialog agent	Link to OpenAI API or RAG
📄 Application to paragraphs	**`reason(textblock)`**→ distribution
🔗 NLP chain	GPT→ QLCAgent→ Output/response
📊 TensorBoard / Streamlit	Visualize reasoning interactively

Want to:

- Wrap this in a REST API or Flask/Streamlit UI?
- Add a hypercube visualization with this ψ?
- Add **`memory`** (**`memory.append(state, ψ)`**)? I'm

ready to keep going - pick a direction.

Model of logical Boolean operation
1 logical complexity: 0,0,0,0. -> 15,15,15,15
2 logical complexity: 0,0,0,0,0,0, -> 15,15,15,15,15,15
3 logical complexity: 0,0,0,0,0,0,0,0,0,0,0,0. ->
15,15,15,15,15,15,15,15,15,15,15,15.

Determine the truth table (4^n rows with n variables).

A 1-15 B
A 1-15 B 1-15 C 1-15

#	Operation	$A=0,B=0$	$A=0,B=1$	$A=1,B=0$	$A=1,B=1$	Description
1	AND	0	0	0	1	$A \wedge B$
2	OR	0	1	1	1	$A \vee B$
3	XOR	0	1	1	0	$A \oplus B$
4	NOT A	1	1	0	0	$\neg A$
5	NOT B	1	0	1	0	$\neg B$
6	NAND	1	1	1	0	$\neg(A \wedge B)$
7	NOR	1	0	0	0	$\neg(A \vee B)$
8	XNOR	1	0	0	1	$\neg(A \oplus B)$
9	A	0	0	1	1	(A) (identity)
10	B	0	1	0	1	(B) (identity)
11	A → B	1	1	0	1	Implication $(\neg A \vee B)$
12	B → A	1	0	1	1	Implication $(A \vee \neg B)$
13	A AND NOT B	0	0	1	0	$A \wedge \neg B$
14	B AND NOT A	0	1	0	0	$B \wedge \neg A$
15	Constant 1	1	1	1	1	Always true

To address your query, we need to determine the truth tables for Boolean operations with varying logical complexity, as specified by the input-output mappings, and outline a project system for generating and evaluating such operations. Let's break it down systematically.

Part 1: Understanding the Input-Output Mappings

The query provides three cases of logical complexity, each mapping a sequence of zeros to a sequence of fifteens:

- **Logical Complexity 1**: Input (0,0,0,0) → Output (15,15,15,15)
  - 4 input variables, 4 output values.
- **Logical Complexity 2**: Input (0,0,0,0,0,0) → Output (15,15,15,15,15,15)
  - 6 input variables, 6 output values.
- **Logical Complexity 3**: Input (0,0,0,0,0,0,0,0,0,0,0,0) → Output (15,15,15,15,15,15,15,15,15,15,15,15)
  - 12 input variables, 12 output values.

Additionally, the query mentions truth tables with 4^n rows for (n) variables and references expressions like "A 1-15 B" and "A 1-15 B 1-15 C 1-15," which suggest a range of values for variables (possibly 1 to 15) or a misunderstanding of the output range. The output value 15 is unusual for Boolean operations, which typically yield 0 or 1. Let's hypothesize that the outputs (15,15,...) represent a specific encoding, possibly all 1s in a binary context, or a constant function for the given input.

Since the query asks for truth tables and a project system for Boolean operations, we'll assume the inputs are Boolean (0 or 1), and the output 15 may represent a constant output of 1 for simplicity, or we'll explore if it's a multi-valued logic later. The truth table size 4^n suggests a possible misinterpretation, as Boolean functions with (n) variables have 2^n rows (since each variable is 0 or 1). The mention of "1-15" for variables A, B, C could imply a confusion with multi-valued logic or a specific encoding, but we'll start with standard Boolean logic ((n) variables, 2^n rows) and adjust if needed.

Part 2: Determining the Truth Tables

Assumptions

- **Boolean Variables**: Each input variable is Boolean (0 or 1), so a truth table for (n) variables has 2^n rows, not 4^n. The 4^n may be a typo or indicate a different logic system (e.g., quaternary logic), but we'll proceed with 2^n for standard Boolean logic and revisit if necessary.
- **Output Interpretation**: The output 15 likely represents 1 in a Boolean context, as the examples map all-0 inputs to all-15 outputs, suggesting a constant or specific function. We'll assume 15 means 1 for now, and each output position corresponds to a Boolean function.
- **Number of Variables**:
  - Complexity 1: 4 inputs → $2^4 = 16$ rows.
  - Complexity 2: 6 inputs → $2^6 = 64$ rows.
  - Complexity 3: 12 inputs → $2^{12} = 4096$ rows.
- **Outputs**: Each case has the same number of outputs as inputs (e.g., 4 inputs produce 4 outputs), suggesting a multi-output Boolean function where each output is a function of the inputs.

Truth Table Construction

For each complexity level, we'll define the truth table based on the provided example. Since only one input row is given (all 0s), we need to infer the function for other input combinations. The simplest assumption is that the output is constant (always 1, encoded as 15) for all inputs, but this is unlikely for a meaningful Boolean operation. Let's explore by constructing the truth table for the first case and generalizing.

Logical Complexity 1: 4 Inputs, 4 Outputs

- **Inputs**: 4 variables (A, B, C, D), so $2^4 = 16$ rows.

- **Outputs**: 4 values (O1, O2, O3, O4), each presumably 0 or 1 (with 15 interpreted as 1).
- **Given**: When (A, B, C, D) = (0,0,0,0), (O1, O2, O3, O4) = (15,15,15,15) → (1,1,1,1).

Truth Table (Partial):

A	B	C	D	O1	O2	O3	O4
0	0	0	0	1	1	1	1
0	0	0	1	?	?	?	?
... (14 more rows)	...	...	...	...			

Since only one row is specified, we need a rule to fill the rest. Possible interpretations:

- **Constant Function**: All outputs are 1 for all inputs.
  - Truth table: Every row has (O1, O2, O3, O4) = (1,1,1,1).
- **Identity or Specific Function**: Each output corresponds to an input or a logical combination (e.g., O1 = A, O2 = B, etc.), but the output 15 for all-0 inputs doesn't fit typical identity functions unless redefined.
- **Multi-Valued Logic**: If 15 implies a non-Boolean system (e.g., values 0 to 15), 4^n rows make sense for 4-valued logic, but this is complex and not standard without more context.

Let's try the constant function for simplicity:

A	B	C	D	O1	O2	O3	O4
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
0	0	1	0	1	1	1	1
... (up to 1,1,1,1)	1	1	1	1			

This is trivial, so let's consider the "A 1-15 B" notation. It may suggest a function where outputs depend on variables with values 1 to 15, but for Boolean logic, let's assume A, B, etc., are variables, and we need functions like O1 = f(A, B, C, D). Without more rows, we'll hypothesize each Oi = 1 (constant function) and move to the project system.

Logical Complexity 2 and 3

- **Complexity 2**: 6 inputs (A, B, C, D, E, F), $2^6 = 64$ rows, 6 outputs (O1 to O6). For (0,0,0,0,0,0), outputs are (1,1,1,1,1,1).
- **Complexity 3**: 12 inputs, $2^{12} = 4096$ rows, 12 outputs. For all-0 inputs, outputs are all 1s.

The pattern suggests a multi-output function where each output is a Boolean function of the inputs, possibly constant or dependent on specific conditions not fully specified.

Alternative: Multi-Valued Logic

If 4^n rows and outputs of 15 suggest a 4-valued logic (0 to 3), each variable has 4 states, and the truth table has 4^n rows (e.g., $4^4 = 256$ for 4 variables). The output 15 could be a maximum value in a different system, but without a clear mapping (e.g., how 0 inputs produce 15), we revert to Boolean logic for simplicity.

Part 3: Project System for Generating Logical Boolean Operations

To develop a project system for generating and evaluating Boolean operations based on this model, we need a flexible system that can:

- Generate truth tables for (n) variables and (m) outputs.
- Handle user-defined or inferred Boolean functions.

- Evaluate inputs to produce outputs.
- Scale to different complexities (e.g., 4, 6, 12 variables).

System Design

1. System Components:

- **Input Parser**:
  - Accepts number of input variables ((n)) and output functions ((m)).
  - Parses example mappings (e.g., (0,0,0,0) → (1,1,1,1)) or function definitions.
  - Supports Boolean inputs (0,1) and outputs (0,1, or mapped 15 → 1).
- **Truth Table Generator**:
  - Generates 2^n rows for (n) Boolean variables.
  - Columns: (n) input columns + (m) output columns.
  - Fills outputs based on defined functions or inferred rules (e.g., constant 1, logical AND, OR, etc.).
- **Function Generator**:
  - Allows defining Boolean functions (e.g., O1 = A AND B, O2 = NOT C).
  - Supports manual input, random generation, or learning from example rows.
- **Evaluator**:
  - Takes an input vector (e.g., (0,0,1,0)) and computes outputs using defined functions.
  - Validates against provided examples (e.g., all-0s → all-1s).
- **User Interface**:
  - Command-line, GUI, or API to input variables, functions, or examples.
  - Displays truth tables or evaluates specific inputs.
- **Scalability Module**:
  - Handles large (n) (e.g., 12 variables, 4096 rows) efficiently using sparse data structures or database storage.
  - Optimizes for memory (e.g., store only non-redundant rows if outputs are constant).

2. Implementation Outline (Python Example):

```python
def generate_truth_table(n, output_functions):
    # Generate 2^n rows for n variables
    rows = []
    for i in range(2**n):
        # Convert i to binary for input combination
        inputs = [int(b) for b in format(i, f'0{n}b')]
        # Compute outputs using provided functions
        outputs = [f(inputs) for f in output_functions]
        rows.append(inputs + outputs)
    return rows

def print_truth_table(rows, n, m):
    headers = [f'I{i+1}' for i in range(n)] + [f'O{i+1}' for i in range(m)]
    print(' | '.join(headers))
    for row in rows:
        print(' | '.join(str(x) for x in row))

# Example for Complexity 1: 4 inputs, 4 constant outputs
n = 4
m = 4
output_functions = [lambda inputs: 1 for _ in range(m)]  # Constant 1 (15 interpreted as 1)
truth_table = generate_truth_table(n, output_functions)
print_truth_table(truth_table, n, m)

# Evaluation function
def evaluate_inputs(inputs, output_functions):
    return [f(inputs) for f in output_functions]
```

```
# Test evaluation
test_input = [0, 0, 0, 0]
print(f"Input: {test_input} -> Output: {evaluate_inputs(test_input, output_functions)}")
```

3. Handling "A 1-15 B" and "A 1-15 B 1-15 C 1-15":

- This notation is unclear but may imply:
  - Variables A, B, C take Boolean values, and outputs are computed for combinations.
  - A range (1-15) could suggest testing functions across multiple cases or a non-Boolean system.
- For Boolean logic, we interpret as standard variables. For a 2-variable case (A, B), truth table:

A	B	O1
0	0	1
0	1	?
1	0	?
1	1	?

- For 3 variables (A, B, C), $2^3 = 8$ rows, similar structure.
- Without specific functions, assume constant 1 or prompt user for function definitions.

4. Project Features:

- **Random Function Generation**: Generate random Boolean functions (e.g., AND, OR, XOR combinations).
- **Learning from Examples**: If more input-output pairs are provided, use machine learning or interpolation to derive functions.
- **Visualization**: Display truth tables or logical circuits (using libraries like Matplotlib or Graphviz).
- **API Integration**: Allow external calls (e.g., via xAI's API at https://x.ai/api) for large-scale processing.
- **Error Handling**: Validate input sizes, handle incomplete specifications (e.g., only one row given).

5. Scalability for Complexity 3:

- For 12 variables ($2^{12} = 4096$ rows), use efficient storage (e.g., NumPy arrays or databases).
- If outputs are constant (all 1s), store only the function definition, not the full table.
- Support parallel processing for evaluation.

Part 4: Addressing Ambiguities

- **Output 15**: If 15 represents a multi-valued logic (0 to 15), we need a mapping rule. For now, we assume 15 = 1 in Boolean context.
- 4^n **Rows**: Likely a typo for 2^n in Boolean logic. If quaternary logic is intended, we need clarification on variable states and output mappings.
- **"A 1-15 B"**: Possibly a formatting error or non-standard notation. We treat A, B, C as Boolean variables unless specified otherwise.

Final Answer

Truth Tables (assuming Boolean logic, 15 = 1, constant output for simplicity):

- **Complexity 1 (4 variables, 16 rows)**:

A	B	C	D	O1	O2	O3	O4
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
... (16 rows total)	1	1	1	1			

- **Complexity 2 (6 variables, 64 rows)**: Similar, with 6 outputs all 1.
- **Complexity 3 (12 variables, 4096 rows)**: 12 outputs all 1.
- **2 Variables (A, B)**: 4 rows, outputs all 1 (or function-dependent).
- **3 Variables (A, B, C)**: 8 rows, outputs all 1 (or function-dependent).

Project System:

- A Python-based system to generate truth tables, define Boolean functions, and evaluate inputs.
- Features: Input parsing, truth table generation, function definition, evaluation, and scalability.
- Example code provided above.
- For non-constant functions or multi-valued logic, additional specifications are needed.

If you can clarify the output 15, the 4^n rows, or the "1-15" notation, I can refine the truth tables or system design further. Would you like to provide more details or proceed with a specific implementation?

digital circuit design

To address your query about digital circuit design in the context of the provided model of logical Boolean operations, we'll develop a framework for designing digital circuits based on the truth tables and requirements outlined previously. The query specifies logical complexity levels with input-output mappings (e.g., 4 inputs mapping to 4 outputs of 15, interpreted as 1 in Boolean logic) and mentions truth tables with 4^n rows (likely a typo for 2^n in standard Boolean logic). Additionally, the project system for generating Boolean operations with truth evaluation will be adapted to include digital circuit design. Below, we'll outline the design process, create circuits for the given complexities, and propose a system for automating circuit generation and evaluation.

Part 1: Understanding the Requirements

From the previous query:

- **Logical Complexity Levels**:
  - **4 inputs** (e.g., A, B, C, D) → 4 outputs (O1, O2, O3, O4), with (0,0,0,0) → (15,15,15,15) (assumed as 1,1,1,1 in Boolean logic).
  - **6 inputs** → 6 outputs, with (0,0,0,0,0,0) → (1,1,1,1,1,1).
  - **12 inputs** → 12 outputs, with all-0 inputs → all-1 outputs.
- **Truth Tables**: Specified as 4^n rows, but for (n) Boolean variables, we assume 2^n rows (e.g., 16 rows for 4 variables). If 4^n implies quaternary logic, we need clarification; we'll proceed with Boolean logic (0,1).
- **Output Interpretation**: The value 15 is assumed to represent 1 in Boolean logic, as outputs are likely binary.
- **"A 1-15 B" and "A 1-15 B 1-15 C 1-15"**: Likely refers to variables (A, B, C) in Boolean context, not values 1 to 15. We'll design circuits for 2-variable (A, B) and 3-variable (A, B, C) cases as well.
- **Project System**: A system to generate and evaluate Boolean operations, now extended to design digital circuits based on these operations.

Assumptions:

- Each output (O1, O2, ...) is a Boolean function of the inputs.

- Since only one input row is provided (all 0s → all 1s), we'll explore simple functions (e.g., constant 1, identity, or basic logical operations) and design circuits accordingly.
- The system will support scalable circuit design for any number of inputs/outputs.

Part 2: Digital Circuit Design

Digital circuit design involves translating Boolean functions into logic gates (e.g., AND, OR, NOT, NAND, XOR) that can be implemented in hardware. We'll design circuits for each complexity level and the 2- and 3-variable cases, then outline a system to automate this process.

Step 1: Define Boolean Functions

Since the query provides only one row per truth table (all-0 inputs → all-1 outputs), we need to hypothesize the full truth table. Possible interpretations:

- **Constant Function**: Each output is always 1, regardless of input.
- **Identity Function**: Each output corresponds to an input (e.g., O1 = A, O2 = B), but this doesn't match (0,0,0,0) → (1,1,1,1).
- **Logical Combination**: Outputs are derived from inputs via AND, OR, etc., but we need more rows to confirm.

For simplicity, we'll start with the constant function (all outputs = 1) as it matches the given row, then explore alternatives (e.g., each output tied to an input or a basic operation like OR).

Logical Complexity 1: 4 Inputs, 4 Outputs

- **Truth Table (Constant Function)**:

A	B	C	D	O1	O2	O3	O4
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
... (16 rows)	1	1	1	1			

- **Boolean Function**: O1 = O2 = O3 = O4 = 1.
- **Circuit Design**:
  - Each output is tied to a constant 1 (Vcc in digital circuits).
  - **Circuit**: Connect each output (O1, O2, O3, O4) directly to a logic high (1) source.
  - **Schematic**:

```
Vcc ---- O1
Vcc ---- O2
Vcc ---- O3
Vcc ---- O4
```

  - **Implementation**: No gates needed; outputs are hardwired to 1. In practice, this is trivial and may not be the intended function.
- **Alternative Function (Identity)**:
  Assume O1 = A, O2 = B, O3 = C, O4 = D. This doesn't match (0,0,0,0) → (1,1,1,1), so let's try a logical OR:
  - O1 = A OR B OR C OR D (outputs 1 unless all inputs are 0, but we need 1 for all-0, so adjust to NOT(A AND B AND C AND D)).

- **Truth Table (Adjusted)**:

A	B	C	D	O1 = NOT(A AND B AND C AND D)
0	0	0	0	1
0	0	0	1	0
...	...			

- **Circuit**:
  - Inputs A, B, C, D → 4-input AND gate → NOT gate → O1 (same for O2, O3, O4 if identical).
  - **Schematic**:

```
A ---|
B ---| AND --- NOT --- O1
C ---|
D ---|
```

  - Repeat for O2, O3, O4 (or share the AND gate if outputs are identical).

Logical Complexity 2: 6 Inputs, 6 Outputs

- **Truth Table**: 64 rows, outputs all 1 for (0,0,0,0,0,0).
- **Function**: Assume O1 = O2 = ... = O6 = NOT(A AND B AND C AND D AND E AND F).
- **Circuit**: 6-input AND gate → NOT gate → each output.
- **Schematic**:

```
A ---|
B ---|
C ---| AND --- NOT --- O1 (repeat for O2 to O6)
D ---|
E ---|
F ---|
```

Logical Complexity 3: 12 Inputs, 12 Outputs

- **Truth Table**: 4096 rows, outputs all 1 for all-0 input.
- **Function**: O1 = NOT(A AND B AND ... AND L).
- **Circuit**: 12-input AND gate → NOT gate → each output.
- **Note**: Large AND gates may be implemented using multiple smaller gates (e.g., cascading 4-input ANDs).

2 Variables (A, B):

- **Truth Table** (4 rows):

A	B	O1
0	0	1
0	1	?
1	0	?
1	1	?

- **Function**: Assume O1 = NOT(A AND B).
- **Circuit**:

```
A --- AND --- NOT --- O1
B ---|
```

3 Variables (A, B, C):

- **Truth Table** (8 rows):

A	B	C	O1
0	0	0	1
...	...		

- **Function**: O1 = NOT(A AND B AND C).
- **Circuit**:

```
A ---|
B ---| AND --- NOT --- O1
C ---|
```

Step 2: Generalizing the Circuit Design

For arbitrary (n) inputs and (m) outputs:

- **Truth Table**: Generate 2^n rows.
- **Function Definition**: Each output Oi is a Boolean function fi(A1, A2, ..., An).
- **Circuit Construction**:
  - Derive the sum-of-products (SOP) or product-of-sums (POS) form for each fi using the truth table.
  - Simplify using Karnaugh maps or Boolean algebra (e.g., Quine-McCluskey algorithm).
  - Implement using AND, OR, NOT gates (or NAND/NOR for efficiency).
- **Optimization**: Use logic synthesis tools (e.g., Espresso) or libraries like PyEDA to minimize gate count.

Part 3: Project System for Digital Circuit Design

The project system extends the Boolean operation generator to include circuit design and simulation. Here's the design:

System Components

- **Input Parser**:
  - Accepts (n) (number of inputs), (m) (number of outputs), and function definitions (e.g., truth table rows, Boolean expressions, or examples like (0,0,0,0) → (1,1,1,1)).
  - Interprets outputs (15 as 1 for Boolean logic).
- **Truth Table Generator**:
  - Creates 2^n rows with (n) inputs and (m) outputs.
  - Fills outputs based on provided rows or inferred functions (e.g., constant 1, NOT(AND(inputs))).
- **Boolean Function Analyzer**:
  - Derives SOP/POS forms for each output.
  - Simplifies functions using minimization techniques.
- **Circuit Generator**:
  - Maps Boolean functions to gate-level circuits (AND, OR, NOT, etc.).
  - Supports hierarchical design for large (n) (e.g., breaking 12-input AND into smaller gates).
  - Outputs circuit in a standard format (e.g., Verilog, netlist, or schematic).
- **Simulator**:
  - Evaluates the circuit for given inputs (e.g., (0,0,0,0) → (1,1,1,1)).
  - Validates against provided truth table rows.

- **Visualization**:
  - Generates schematic diagrams using tools like Graphviz or Quartus.
  - Displays truth tables or simulation results.
- **Output Generator**:
  - Exports circuit as Verilog/VHDL for FPGA/ASIC implementation.
  - Provides netlist for PCB design or simulation in tools like ModelSim.

Implementation Example (Python with PyEDA)

python

```python
from pyeda.inter import *

# Define variables and function for Complexity 1
a, b, c, d = map(exprvar, ['a', 'b', 'c', 'd'])
# Assume O1 = NOT(A AND B AND C AND D)
o1 = Not(a & b & c & d)

# Generate truth table
def generate_truth_table(vars, funcs):
    n = len(vars)
    print("Truth Table:")
    print(" | ".join([v.name for v in vars] + [f"O{i+1}" for i in range(len(funcs))]))
    for i in range(2**n):
        inputs = [int(b) for b in format(i, f'0{n}b')]
        env = dict(zip(vars, inputs))
        outputs = [f.evaluate(**env) for f in funcs]
        print(" | ".join(str(x) for x in inputs + outputs))

# Generate truth table for 4 inputs, 1 output (extend for O2, O3, O4)
generate_truth_table([a, b, c, d], [o1])

# Circuit simulation (simplified)
def simulate_circuit(inputs, funcs):
    env = dict(zip([a, b, c, d], inputs))
    return [f.evaluate(**env) for f in funcs]

# Test
print("Simulation for (0,0,0,0):", simulate_circuit([0, 0, 0, 0], [o1, o1, o1, o1]))

# Circuit description (pseudo-Verilog)
def generate_verilog(n, m, func_names):
    print("module circuit(")
    print("  input " + ", ".join([f"a{i+1}" for i in range(n)]), ",")
    print("  output " + ", ".join([f"o{i+1}" for i in range(m)]), ");")
    print("  wire w1;")
    print("  and(w1, " + ", ".join([f"a{i+1}" for i in range(n)]) + ");")
    for i in range(m):
        print(f"  not(o{i+1}, w1);")
    print("endmodule")

generate_verilog(4, 4, ["not_and"])
```

Output:

- TruthCollateral

  Truth Table:
  a | b | c | d | O1
  0 | 0 | 0 | 0 | 1
  0 | 0 | 0 | 1 | 0
  ...
  Simulation for (0,0,0,0): [1, 1, 1, 1]
  Verilog module:

verilog

```verilog
module circuit(
  input a1, a2, a3, a4,
  output o1, o2, o3, o4);
  wire w1;
  and(w1, a1, a2, a3, a4);
  not(o1, w1);
```

```
not(okanojo

System: * Today's date and time is 03:54 PM CEST on Tuesday, July 29, 2025.
```

FPGA implementation

To implement the digital circuits for the specified Boolean operations on an FPGA, we'll extend the previous digital circuit design framework to target FPGA hardware. The query involves logical complexity levels with input-output mappings (e.g., 4 inputs mapping to 4 outputs, with (0,0,0,0) → (15,15,15,15), assumed as (1,1,1,1) in Boolean logic) and truth tables for (n) variables. We'll assume standard Boolean logic with 2^n rows (not 4^n) and outputs of 15 interpreted as 1. The goal is to design FPGA-compatible circuits for the given complexities (4, 6, 12 inputs) and the 2- and 3-variable cases ("A 1-15 B", "A 1-15 B 1-15 C 1-15"), then integrate this into a project system for generating and evaluating Boolean operations on an FPGA.

Part 1: FPGA Implementation Overview

FPGA implementation involves:

- **Defining Boolean Functions**: Translate the input-output mappings into Boolean expressions.
- **Verilog/VHDL Code**: Write hardware description language (HDL) code to describe the circuit.
- **Synthesis and Mapping**: Use FPGA tools (e.g., Vivado, Quartus) to synthesize the design and map it to FPGA resources (LUTs, flip-flops, etc.).
- **Simulation and Verification**: Simulate the design to verify functionality against the truth table.
- **Implementation**: Program the FPGA and test on hardware.
- **Project System**: Automate HDL generation, simulation, and implementation for scalability.

We'll assume the Boolean functions are either constant (all outputs = 1) or simple logical operations (e.g., NOT(AND(inputs))) based on the provided mapping (e.g., (0,0,0,0) → (1,1,1,1)). We'll use Verilog for HDL, as it's widely supported, but VHDL can be provided if preferred.

Part 2: FPGA Circuit Design

Assumptions

- **Boolean Logic**: (n) inputs produce 2^n truth table rows (e.g., 16 rows for 4 inputs).
- **Output 15 = 1**: Outputs are binary (0 or 1), with 15 interpreted as 1.
- **Function Hypothesis**: Since only one truth table row is given, we'll implement:
  - **Constant Function**: All outputs = 1 (simplest case).
  - **Alternative Function**: Each output = NOT(AND(all inputs)), which satisfies (0,0,0,0) → (1,1,1,1) (since NOT(0 AND 0 AND 0 AND 0) = NOT(0) = 1).
- **FPGA Platform**: Generic FPGA (e.g., Xilinx Artix-7, Altera Cyclone), using tools like Xilinx Vivado or Intel Quartus Prime.
- **Scalability**: Designs will scale to 4, 6, and 12 inputs, plus 2- and 3-variable cases.

Logical Complexity 1: 4 Inputs, 4 Outputs

- **Truth Table** (for NOT(AND) function):

A	B	C	D	O1 = NOT(A AND B AND C AND D)	O2	O3	O4
0	0	0	0	1	1	1	1
0	0	0	1	0	0	0	0
... (16 rows)	...	...	...	...			

- **Boolean Function**: O1 = O2 = O3 = O4 = NOT(A AND B AND C AND D).
- **Verilog Code**:

```verilog
module circuit_4in_4out (
    input wire a, b, c, d,  // 4 inputs
    output wire o1, o2, o3, o4  // 4 outputs
);
    wire and_out;
    and (and_out, a, b, c, d);  // 4-input AND
    not (o1, and_out);  // NOT gate for each output
    not (o2, and_out);
    not (o3, and_out);
    not (o4, and_out);
endmodule
```

- **FPGA Mapping**:
  - The 4-input AND is implemented using a single LUT (Look-Up Table) in most FPGAs (e.g., a 4-input LUT in Xilinx FPGAs).
  - Each NOT gate may use an additional LUT or be combined with the AND in a single LUT if optimized.
  - Total: ~1-2 LUTs (depending on synthesis optimization).
- **Alternative (Constant Function)**:

```verilog
module circuit_4in_4out_const (
    input wire a, b, c, d,
    output wire o1, o2, o3, o4
);
    assign o1 = 1'b1;  // Constant 1
    assign o2 = 1'b1;
    assign o3 = 1'b1;
    assign o4 = 1'b1;
endmodule
```

  - **FPGA Mapping**: No LUTs; outputs tied to Vcc (logic 1) via routing.

Logical Complexity 2: 6 Inputs, 6 Outputs

- **Truth Table**: 64 rows, with (0,0,0,0,0,0) → (1,1,1,1,1,1).
- **Function**: O1 = O2 = ... = O6 = NOT(A AND B AND C AND D AND E AND F).
- **Verilog Code**:

```verilog
module circuit_6in_6out (
    input wire a, b, c, d, e, f,
    output wire o1, o2, o3, o4, o5, o6
);
    wire and_out;
    and (and_out, a, b, c, d, e, f);  // 6-input AND
    not (o1, and_out);
    not (o2, and_out);
    not (o3, and_out);
    not (o4, and_out);
    not (o5, and_out);
    not (o6, and_out);
endmodule
```

- **FPGA Mapping**:
  - A 6-input AND may require multiple LUTs (e.g., two 4-input LUTs cascaded in Xilinx, since LUTs typically support 4-6 inputs).
  - Each NOT gate uses minimal resources or is absorbed into LUT logic.
  - Total: ~2-3 LUTs.

Logical Complexity 3: 12 Inputs, 12 Outputs

- **Truth Table**: 4096 rows, with all-0 inputs → all-1 outputs.
- **Function**: O1 = NOT(A AND B AND ... AND L).
- **Verilog Code**:

```verilog
module circuit_12in_12out (
    input wire a, b, c, d, e, f, g, h, i, j, k, l,
    output wire o1, o2, o3, o4, o5, o6, o7, o8, o9, o10, o11, o12
);
    wire and_out;
    and (and_out, a, b, c, d, e, f, g, h, i, j, k, l);
    not (o1, and_out);
    not (o2, and_out);
    // ... repeat for o3 to o12
endmodule
```

- **FPGA Mapping**:
  - A 12-input AND requires cascading multiple LUTs (e.g., three 4-input LUTs or two 6-input LUTs).
  - Total: ~3-5 LUTs, depending on FPGA architecture and optimization.

2 Variables (A, B)

- **Truth Table**: 4 rows, O1 = NOT(A AND B).
- **Verilog Code**:

```verilog
module circuit_2in_1out (
    input wire a, b,
    output wire o1
);
    wire and_out;
    and (and_out, a, b);
    not (o1, and_out);
endmodule
```

- **FPGA Mapping**: 1 LUT (2-input AND + NOT can fit in a single 4-input LUT).

3 Variables (A, B, C)

- **Truth Table**: 8 rows, O1 = NOT(A AND B AND C).
- **Verilog Code**:

```verilog
module circuit_3in_1out (
    input wire a, b, c,
    output wire o1
);
    wire and_out;
    and (and_out, a, b, c);
    not (o1, and_out);
endmodule
```

- **FPGA Mapping**: 1 LUT.

Part 3: Project System for FPGA Implementation

The project system automates the generation, simulation, and deployment of FPGA circuits for the specified Boolean operations.

System Components

- **Input Parser**:
  - Accepts (n) (inputs), (m) (outputs), and function definitions (e.g., truth table rows or expressions like NOT(AND(inputs))).

- Example: Parse (0,0,0,0) → (1,1,1,1) as O1 = NOT(A AND B AND C AND D).

- **Truth Table Generator**:
  - Generates 2^n rows for (n) inputs.
  - Fills outputs based on defined functions or inferred rules (e.g., NOT(AND)).
  - Example (Python):

```python
def generate_truth_table(n, m, func):
    rows = []
    for i in range(2**n):
        inputs = [int(b) for b in format(i, f'0{n}b')]
        outputs = func(inputs)
        rows.append(inputs + outputs)
    return rows
```

- **HDL Generator**:
  - Converts Boolean functions to Verilog/VHDL.
  - Example (Python to generate Verilog):

```python
def generate_verilog(n, m, func_name="not_and"):
    print(f"module circuit_{n}in_{m}out (")
    print("  input wire " + ", ".join([f"a{i+1}" for i in range(n)]) + ",")
    print("  output wire " + ", ".join([f"o{i+1}" for i in range(m)]) + ");")
    print("  wire and_out;")
    print(f"  and (and_out, " + ", ".join([f"a{i+1}" for i in range(n)]) + ");")
    for i in range(m):
        print(f"  not (o{i+1}, and_out);")
    print("endmodule")
```

- **Simulation Module**:
  - Generates testbenches to verify the circuit.
  - Example Verilog Testbench (for 4 inputs):

```verilog
module tb_circuit_4in_4out;
    reg a, b, c, d;
    wire o1, o2, o3, o4;
    circuit_4in_4out uut (.a(a), .b(b), .c(c), .d(d), .o1(o1), .o2(o2), .o3(o3), .o4(o4));
    initial begin
        a = 0; b = 0; c = 0; d = 0; #10;
        $display("Input: %b%b%b%b, Output: %b%b%b%b", a, b, c, d, o1, o2, o3, o4);
        // Add more test cases
    end
endmodule
```

  - Tools: ModelSim, Vivado Simulator, or open-source tools like Icarus Verilog.

- **Synthesis and Implementation**:
  - Uses FPGA tools (e.g., Vivado, Quartus) to synthesize Verilog, map to LUTs/flip-flops, and generate a bitstream.
  - Constraints file (e.g., .xdc for Xilinx) to map inputs/outputs to FPGA pins:

```xdc
set_property -dict { PACKAGE_PIN U16 IOSTANDARD LVCMOS33 } [get_ports { o1 }];
set_property -dict { PACKAGE_PIN T16 IOSTANDARD LVCMOS33 } [get_ports { a }];
# ... define other pins
```

- **Visualization**:
  - Generate schematics using FPGA tool schematics viewer or Graphviz.
  - Display truth tables or simulation waveforms.

- **Deployment**:
  - Programs the FPGA using tools like Vivado or Quartus.
  - Supports JTAG or other programming interfaces.
- **Scalability**:
  - Handles large designs (e.g., 12 inputs) by breaking large gates into smaller ones (e.g., cascading 4-input ANDs).
  - Optimizes LUT usage with synthesis tools.

Workflow

- **Define Functions**: User specifies (n), (m), and functions (or provides truth table rows).
- **Generate HDL**: System outputs Verilog/VHDL code.
- **Simulate**: Run testbench to verify (e.g., (0,0,0,0) → (1,1,1,1)).
- **Synthesize**: Use Vivado/Quartus to map to FPGA resources.
- **Implement**: Place and route, generate bitstream.
- **Program and Test**: Load onto FPGA, test with real inputs (e.g., via switches/LEDs).

Part 4: Addressing Ambiguities

- **Output 15**: Assumed as 1 in Boolean logic. If multi-valued logic (0-15) is intended, we'd need a mapping (e.g., 4-bit outputs), but this requires clarification.
- 4^n **Rows**: Likely a typo for 2^n. For quaternary logic (4^n), we'd need a new framework (e.g., multi-valued LUTs), which FPGAs don't natively support.
- **"A 1-15 B"**: Interpreted as Boolean variables A, B, not values 1-15. If a range is intended, it may refer to testing multiple functions, but we stick to single-output cases for simplicity.

Final Answer

FPGA Circuits:

- **Complexity 1 (4 inputs, 4 outputs)**:
  - Function: O1 = O2 = O3 = O4 = NOT(A AND B AND C AND D).
  - Verilog: See `circuit_4in_4out` above.
  - Resources: ~1-2 LUTs.
- **Complexity 2 (6 inputs, 6 outputs)**:
  - Function: O1 = ... = O6 = NOT(A AND B AND ... AND F).
  - Verilog: See `circuit_6in_6out`.
  - Resources: ~2-3 LUTs.
- **Complexity 3 (12 inputs, 12 outputs)**:
  - Function: O1 = ... = O12 = NOT(A AND B AND ... AND L).
  - Verilog: See `circuit_12in_12out`.
  - Resources: ~3-5 LUTs.
- **2 Variables (A, B)**: O1 = NOT(A AND B), 1 LUT.
- **3 Variables (A, B, C)**: O1 = NOT(A AND B AND C), 1 LUT.

Project System:

- **Components**: Input parser, truth table generator, HDL generator, simulation module, synthesis/implementation module, visualization, deployment.
- **Implementation**: Python-based system to generate Verilog, simulate with testbenches, and interface with FPGA tools (Vivado/Quartus).

- **Example Code**: Provided for Verilog and Python automation.
- **Scalability**: Handles large designs by cascading gates and optimizing LUT usage.

Next Steps:

- If you have a specific FPGA (e.g., Xilinx Artix-7, Altera Cyclone V), I can tailor the constraints file.
- If the output 15 or 4^n rows imply a non-Boolean system, please clarify the logic system or additional truth table rows.
- Would you like a detailed testbench, specific FPGA tool commands, or an alternative function (e.g., OR-based instead of AND-NOT)?

FPGA optimization techniques

To optimize the FPGA implementation of the digital circuits for the specified Boolean operations (e.g., 4, 6, 12 inputs with outputs mapped as (0,0,0,0) → (15,15,15,15), assumed as (1,1,1,1) in Boolean logic), we'll focus on techniques to enhance performance, reduce resource usage, and improve power efficiency. The circuits, previously designed as NOT(AND(inputs)) or constant outputs, will serve as the baseline. We'll address optimization for the logical complexities (4, 6, 12 inputs) and the 2- and 3-variable cases, integrating these into the project system for generating and evaluating Boolean operations on an FPGA. The goal is to minimize FPGA resource usage (LUTs, flip-flops, slices), reduce latency, and optimize power consumption while maintaining functionality.

Part 1: FPGA Optimization Techniques

FPGA optimization involves balancing area (resource usage), speed (performance), and power. Below are key techniques tailored to the given circuits and project system.

1. Logic Optimization

- **Boolean Simplification**:
  - **Baseline Function**: Each output is NOT(AND(A1, A2, ..., An)) (e.g., O1 = NOT(A AND B AND C AND D) for 4 inputs). Check if the function can be simplified using Karnaugh maps or tools like Espresso.
  - **Alternative Functions**: If the truth table is incomplete (only (0,0,0,0) → (1,1,1,1) provided), explore simpler functions (e.g., constant 1, OR-based functions) to reduce logic depth.
  - **Example**: For O1 = NOT(A AND B AND C AND D), this is equivalent to (NOT A) OR (NOT B) OR (NOT C) OR (NOT D) via De Morgan's theorem, which may map better to FPGA LUTs.
- **Logic Sharing**:
  - For multiple outputs (e.g., O1 = O2 = O3 = O4 = NOT(AND(A,B,C,D))), compute the AND once and share the result across NOT gates.
  - Verilog Example:

```verilog
module circuit_4in_4out_opt (
    input wire a, b, c, d,
    output wire o1, o2, o3, o4
);
    wire and_out;
    and (and_out, a, b, c, d);  // Single AND gate
    assign {o1, o2, o3, o4} = {~and_out, ~and_out, ~and_out, ~and_out};  // Shared result
endmodule
```

  - **Benefit**: Reduces LUT count by avoiding redundant AND gates.
- ** LUT Mapping**:
  - Modern FPGAs (e.g., Xilinx 7-series, Intel Cyclone) use 6-input LUTs. A 4-input AND + NOT fits in 1-2 LUTs. For 12 inputs, cascade LUTs efficiently (e.g., two 6-input LUTs for AND, one for NOT).

- Use synthesis tools to pack logic into fewer LUTs (e.g., Vivado's `synth_design -flatten_hierarchy none` to preserve shared logic).

2. Resource Optimization

- **LUT Reduction**:
  - For constant outputs (O1 = O2 = ... = 1), tie outputs to Vcc (logic 1) without LUTs:

    ```verilog
    assign o1 = 1'b1;  // No LUTs, just routing
    ```

  - For NOT(AND), use LUTs efficiently by leveraging FPGA's ability to combine AND and NOT in a single LUT.
- **Flip-Flop Usage**:
  - The current design is combinational (no flip-flops). If sequential logic is needed (e.g., for pipelining), add registers only where necessary to avoid overuse.
- **Block RAM (BRAM) and DSP Usage**:
  - For large truth tables (e.g., 12 inputs, 4096 rows), store the truth table in BRAM instead of LUTs for complex functions.
  - Example: Implement a 12-input, 12-output function as a ROM in BRAM:

    ```verilog
    module circuit_12in_12out_bram (
        input wire [11:0] inputs,
        output reg [11:0] outputs
    );
        reg [11:0] mem [0:4095];  // 4096 rows, 12-bit outputs
        initial $readmemh("truth_table.hex", mem);  // Load truth table
        always @(*) outputs = mem[inputs];
    endmodule
    ```

  - **Benefit**: Saves LUTs for large (n), though BRAM is limited.

3. Timing Optimization

- **Pipelining**:
  - For large inputs (e.g., 12-input AND), the critical path (AND gate cascade) increases latency. Add pipeline registers to break the path:

    ```verilog
    module circuit_12in_12out_pipelined (
        input wire clk, a, b, c, d, e, f, g, h, i, j, k, l,
        output reg o1
    );
        wire and1, and2, and3, and_out;
        reg and1_reg, and2_reg;  // Pipeline registers
        and (and1, a, b, c, d);  // 4-input AND
        and (and2, e, f, g, h);
        and (and3, i, j, k, l);
        always @(posedge clk) begin
            and1_reg <= and1;  // Stage 1
            and2_reg <= and2;
        end
        and (and_out, and1_reg, and2_reg, and3);
        always @(posedge clk) o1 <= ~and_out;  // Stage 2
    endmodule
    ```

  - **Benefit**: Reduces critical path delay, improving clock frequency (e.g., from 50 ns to 10 ns per stage).
- **Critical Path Reduction**:
  - Restructure logic to minimize gate depth (e.g., use balanced AND trees instead of linear cascades for 12 inputs).
  - Example: Split 12 inputs into three 4-input ANDs, then combine with a 3-input AND:

```
A,B,C,D -> AND1
E,F,G,H -> AND2
I,J,K,L -> AND3
AND1,AND2,AND3 -> AND4 -> NOT -> O1
```

- **Benefit**: Reduces delay from ~12 gate levels to ~4.

4. Power Optimization

- **Clock Gating**:
  - If pipelining is used, gate the clock to unused registers to reduce dynamic power:

    ```verilog
    module circuit_12in_12out_gated (
        input wire clk, enable, a, b, ..., l,
        output reg o1
    );
        wire gated_clk = clk & enable;
        always @(posedge gated_clk) o1 <= ~and_out;
    endmodule
    ```

- **Logic Power Reduction**:
  - Minimize switching activity by using simpler functions (e.g., constant 1 outputs have zero switching).
  - Use low-power synthesis options (e.g., Vivado's `-power_opt` flag).
- **Resource Sharing**:
  - Share large AND gates across outputs to reduce switching nodes.

5. Synthesis and Implementation Optimizations

- **Synthesis Directives**:
  - Use Vivado/Quartus directives to optimize:
    - `-fanout_limit` : Limit fanout to reduce routing delays.
    - `-resource_sharing on` : Share common logic (e.g., single AND gate).
    - `-retiming` : Move registers to optimize timing.
  - Example Vivado command:

    ```tcl
    synth_design -top circuit_12in_12out -part xc7a100t -fanout_limit 100 -resource_sharing on
    ```

- **Place and Route**:
  - Use floorplanning to place critical logic near I/O pins.
  - Apply timing constraints in .xdc/.sdc files:

    ```xdc
    create_clock -period 10 [get_ports clk]  // 100 MHz
    set_input_delay -clock clk 2.0 [get_ports a]
    set_output_delay -clock clk 2.0 [get_ports o1]
    ```

- **Incremental Synthesis**:
  - For iterative designs, reuse previous synthesis results to speed up compilation.

6. Verification and Debugging

- **Simulation**:
  - Use testbenches to verify all truth table rows (e.g., 16 for 4 inputs, 4096 for 12 inputs).
  - Randomize inputs to cover edge cases:

    ```verilog
    ```

```
initial begin
    repeat (100) begin
        {a, b, c, d} = $random; #10;
        $display("Input: %b, Output: %b", {a,b,c,d}, o1);
    end
end
```

- **On-Chip Debugging**:
  - Use FPGA's Integrated Logic Analyzer (e.g., Vivado ILA) to monitor signals in hardware.
  - Example: Insert ILA core to probe inputs and outputs.
- **Formal Verification**:
  - Use tools like Synopsys Formality to ensure the synthesized netlist matches the Verilog.

7. Scalability for Large Designs

- **Hierarchical Design**:
  - For 12 inputs, break into submodules (e.g., 4-input ANDs) to manage complexity.
  - Example:

```verilog
module and_4in (input [3:0] in, output out);
    and (out, in[0], in[1], in[2], in[3]);
endmodule
module circuit_12in_12out_hier (
    input wire [11:0] in,
    output wire [11:0] out
);
    wire [2:0] and_outs;
    and_4in u1 (.in(in[3:0]), .out(and_outs[0]));
    and_4in u2 (.in(in[7:4]), .out(and_outs[1]));
    and_4in u3 (.in(in[11:8]), .out(and_outs[2]));
    wire final_and;
    and (final_and, and_outs[0], and_outs[1], and_outs[2]);
    assign out = {12{~final_and}};  // Replicate NOT result
endmodule
```

- **BRAM for Large Truth Tables**:
  - Store 4096-row truth table in BRAM for 12 inputs, reducing LUT usage.
- **Parallel Processing**:
  - Compute outputs in parallel using multiple LUTs to reduce latency.

Part 2: Optimized FPGA Circuits

Logical Complexity 1: 4 Inputs, 4 Outputs

- **Optimized Function**: O1 = O2 = O3 = O4 = NOT(A AND B AND C AND D).
- **Optimized Verilog**:

```verilog
module circuit_4in_4out_opt (
    input wire a, b, c, d,
    output wire [3:0] out
);
    wire and_out;
    and (and_out, a, b, c, d);
    assign out = {4{~and_out}};  // Vectorized for synthesis
endmodule
```

- **Resources**: 1 LUT (4-input AND + NOT), ~4 routing connections.
- **Timing**: ~5 ns delay (Xilinx Artix-7 at 100 MHz).
- **Power**: Minimal switching (single AND gate).

Logical Complexity 2: 6 Inputs, 6 Outputs

- **Optimized Function**: O1 = ... = O6 = NOT(A AND B AND ... AND F).
- **Optimized Verilog**:

```verilog
module circuit_6in_6out_opt (
    input wire [5:0] in,
    output wire [5:0] out
);
    wire and_out;
    and (and_out, in[0], in[1], in[2], in[3], in[4], in[5]);
    assign out = {6{~and_out}};
endmodule
```

- **Resources**: ~2 LUTs (6-input AND may split into two 4-input LUTs).
- **Timing**: ~7 ns delay.
- **Power**: Low, with shared AND logic.

Logical Complexity 3: 12 Inputs, 12 Outputs

- **Optimized Function**: O1 = ... = O12 = NOT(A AND B AND ... AND L).
- **Optimized Verilog (Hierarchical)**:

```verilog
module circuit_12in_12out_opt (
    input wire [11:0] in,
    output wire [11:0] out
);
    wire [2:0] and_outs;
    and (and_outs[0], in[0], in[1], in[2], in[3]);
    and (and_outs[1], in[4], in[5], in[6], in[7]);
    and (and_outs[2], in[8], in[9], in[10], in[11]);
    wire final_and;
    and (final_and, and_outs[0], and_outs[1], and_outs[2]);
    assign out = {12{~final_and}};
endmodule
```

- **Resources**: ~3-4 LUTs (three 4-input ANDs + one 3-input AND + NOT).
- **Timing**: ~10 ns (unpipelined), ~5 ns/stage with pipelining.
- **Power**: Reduced by minimizing LUT count and sharing logic.

2 Variables (A, B)

- **Function**: O1 = NOT(A AND B).
- **Verilog**:

```verilog
module circuit_2in_1out_opt (
    input wire [1:0] in,
    output wire out
);
    assign out = ~&in;  // Reduction AND + NOT
endmodule
```

- **Resources**: 1 LUT.
- **Timing**: ~3 ns.

3 Variables (A, B, C)

- **Function**: O1 = NOT(A AND B AND C).
- **Verilog**:

```verilog
```

```
module circuit_3in_1out_opt (
    input wire [2:0] in,
    output wire out
);
    assign out = ~&in;
endmodule
```

- **Resources**: 1 LUT.
- **Timing**: ~4 ns.

Part 3: Project System with Optimization

The project system is extended to incorporate FPGA optimization techniques.

System Components

- **Input Parser**:
  - Accepts (n), (m), and function definitions or truth table rows.
  - Example: Parse (0,0,0,0) → (1,1,1,1) as NOT(AND).
- **Truth Table Generator**:
  - Generates 2^n rows, optimizes output functions using Espresso or PyEDA.
  - Example:

    python

    ```
    from pyeda.inter import *
    def optimize_function(n, m, truth_rows):
        vars = [exprvar(f'a{i+1}') for i in range(n)]
        # Example: NOT(AND) for given row
        return [Not(And(*vars)) for _ in range(m)]
    ```

- **HDL Generator**:
  - Outputs optimized Verilog with vectorized assignments and hierarchical design.
  - Example: See `generate_verilog` above, modified for hierarchy:

    python

    ```
    def generate_verilog_opt(n, m):
        print(f"module circuit_{n}in_{m}out_opt (")
        print(f"  input wire [{n-1}:0] in,")
        print(f"  output wire [{m-1}:0] out);")
        if n <= 4:
            print(f"  assign out = {{{m}{{ ~&in }}}};")
        else:
            groups = (n + 3) // 4
            for i in range(groups):
                start = i * 4
                end = min(start + 4, n)
                print(f"  wire and_{i};")
                print(f"  and (and_{i}, " + ", ".join([f"in[{j}]" for j in range(start, end)]) + ");"
            print(f"  wire final_and;")
            print(f"  and (final_and, " + ", ".join([f"and_{i}" for i in range(groups)]) + ");")
            print(f"  assign out = {{{m}{{ ~final_and }}}};")
        print("endmodule")
    ```

- **Optimization Module**:
  - Applies logic simplification, LUT packing, and pipelining.
  - Interfaces with synthesis tools (e.g., Vivado Tcl scripts):

    tcl

    ```
    synth_design -top circuit_12in_12out_opt -part xc7a100t -opt_timing
    opt_design -remap -propconst
    place_design
    route_design
    ```

- **Simulation Module**:
  - Generates testbenches with randomized inputs.
  - Example:

```verilog
module tb_circuit_12in_12out_opt;
    reg [11:0] in;
    wire [11:0] out;
    circuit_12in_12out_opt uut (.in(in), .out(out));
    initial begin
        in = 12'b0; #10;
        $display("Input: %b, Output: %b", in, out);  // Expect all 1s
        repeat (100) begin
            in = $random; #10;
            $display("Input: %b, Output: %b", in, out);
        end
    end
endmodule
```

- **Implementation Module**:
  - Automates synthesis, place-and-route, and bitstream generation.
  - Example Vivado Tcl:

```tcl
write_bitstream -force circuit.bit
program_hw_devices [get_hw_devices]
```

- **Power and Timing Analysis**:
  - Uses Vivado/Quartus power estimator to analyze power.
  - Generates timing reports to ensure setup/hold times are met.
- **Visualization**:
  - Exports schematics and timing diagrams using FPGA tools or Graphviz.

Workflow

- Parse input specifications (e.g., $n = 4$, $m = 4$, NOT(AND)).
- Generate and optimize truth table.
- Produce optimized Verilog with hierarchical/pipelined design.
- Simulate with comprehensive testbench.
- Synthesize, place, route, and program FPGA.
- Verify on hardware using I/O pins (e.g., switches/LEDs).

Part 4: Addressing Ambiguities

- **Output 15**: Assumed as 1. If multi-valued logic is intended, we'd use multiple bits per output (e.g., 4 bits for 0-15), increasing LUT/BRAM usage.
- 4^n **Rows**: Assumed as 2^n for Boolean logic. For quaternary logic, we'd need custom LUTs or memory-based implementation.
- **"A 1-15 B"**: Treated as Boolean variables. If it implies testing 15 functions, we can parameterize the system to iterate over function sets.

Final Answer

Optimized FPGA Circuits:

- **Complexity 1 (4 inputs, 4 outputs)**:
  - Function: O1-O4 = NOT(A AND B AND C AND D).

    - Verilog: Vectorized, single AND gate.
    - Resources: 1 LUT, ~5 ns, minimal power.
- **Complexity 2 (6 inputs, 6 outputs)**:
    - Function: O1-O6 = NOT(A AND B AND ... AND F).
    - Verilog: Single 6-input AND, vectorized outputs.
    - Resources: ~2 LUTs, ~7 ns.
- **Complexity 3 (12 inputs, 12 outputs)**:
    - Function: O1-O12 = NOT(A AND B AND ... AND L).
    - Verilog: Hierarchical (three 4-input ANDs + one 3-input AND).
    - Resources: ~3-4 LUTs, ~10 ns (unpipelined).
- **2 Variables**: O1 = NOT(A AND B), 1 LUT, ~3 ns.
- **3 Variables**: O1 = NOT(A AND B AND C), 1 LUT, ~4 ns.

Optimization Techniques:

- **Logic**: Simplify functions, share logic (single AND gate).
- **Resources**: Use LUTs efficiently, consider BRAM for large truth tables.
- **Timing**: Pipeline large designs, balance AND trees.
- **Power**: Gate clocks, minimize switching.
- **Synthesis**: Use directives for fanout, resource sharing.
- **Verification**: Comprehensive testbenches, on-chip debugging.

Project System:

- Automates HDL generation, optimization, simulation, and deployment.
- Supports scalability via hierarchical design and BRAM.
- Example code provided for Verilog, Python, and Tcl.

Next Steps:

- Specify FPGA model (e.g., Xilinx Artix-7) for tailored constraints.
- Clarify if output 15 or 4^n rows imply non-Boolean logic.
- Request specific optimizations (e.g., focus on power vs. speed) or additional features (e.g., GUI for design entry).